AFRL-SN-RS-TR-2001-40
Final Technical Report
March 2001

APPLICATION-DRIVEN RELIABILITY MEASURES AND EVALUATION TOOL FOR FAULT-TOLERANT REAL-TIME SYSTEMS

University of Massachusetts

Sponsored by
Defense Advanced Research Projects Agency
DARPA Order No. E349

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

AIR FORCE RESEARCH LABORATORY
SENSORS DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK

This report has been reviewed by the Air Force Research Laboratory, Information Directorate, Public Affairs Office (IFOIPA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

AFRL-SN-RS-TR-2001-40 has been reviewed and is approved for publication.

APPROVED: R. Kohler

RALPH KOHLER
Project Engineer

FOR THE DIRECTOR: Robert G. Polce

ROBERT G. POLCE, Chief
Rome Operations Office
Sensors Directorate

If your address has changed or if you wish to be removed from the Air Force Research Laboratory Rome Research Site mailing list, or if the addressee is no longer employed by your organization, please notify AFRL/SNRT, 26 Electronic Pky, Rome, NY 13441-4514. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notices on a specific document require that it be returned.

APPLICATION-DRIVEN RELIABILITY MEASURES AND EVALUATION TOOL FOR FAULT-TOLERANT REAL-TIME SYSTEMS

C. M. Krishna
and I. Koren

Contractor: University of Massachusetts
Contract Number: F30602-96-1-0341
Effective Date of Contract: 23 August 1996
Contract Expiration Date: 22 August 2000
Short Title of Work: Application-Driven Reliability Measures and Evaluation Tool For Fault-Tolerant Real-Time Systems
Period of Work Covered: Aug 96 - Aug 00

Principal Investigator: C. M. Krishna
Phone: (413) 545-0766
AFRL Project Engineer: Ralph Kohler
Phone: (315) 330-2016

This research was supported by the Defense Advanced Research Projects Agency of the Department of Defense and was monitored by Ralph Kohler, AFRL/SNRT, 26 Electronic Pky, Rome, NY.

REPORT DOCUMENTATION PAGE	Form Approved OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	MARCH 2001	Final Oct 96 - Aug 00

4. TITLE AND SUBTITLE
APPLICATION-DRIVEN RELIABILITY MEASURES AND EVALUATION TOOL FOR FAULT-TOLERANT REAL-TIME SYSTEMS

5. FUNDING NUMBERS
C - F30602-96-1-0341
PE - 62301E
PR - D985
TA - 00
WU - P1

6. AUTHOR(S)
C. M. Krishna and I. Koren

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
University of Massachusetts
Department of Electrical and Computer Engineering
Amherst MA 01003

8. PERFORMING ORGANIZATION REPORT NUMBER
N/A

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Defense Advanced Research Projects Agency
3701 North Fairfax Drive
Arlington VA 22203

Air Force Research Laboratory/SNRT
26 Electronic Pky
Rome NY 13441-4514

10. SPONSORING/MONITORING AGENCY REPORT NUMBER
AFRL-SN-RS-TR-2001-40

11. SUPPLEMENTARY NOTES
Air Force Research Laboratory Project Engineer: Ralph Kohler/SNRT/(315) 330-2016

12a. DISTRIBUTION AVAILABILITY STATEMENT
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

The goals of this project were the development of performance measures suitable for use in real-time embedded systems. Developed a network measure to guide the designer in choosing an appropriate network topology. The measure combines graphic-theoretic concepts in evaluating the underlying reliability of the network and other means to evaluate the ability of the network to support interprocessor traffic. A second measure, called the computer measure, is meant to evaluated the computer system in terms that are of importance for real-time applications. A simulator testbed called TRIDENT was built to evaluate the network and surge measures. Much of the technology developed under this project is being transferred to the Jet Propulsion Laboratory (JPL).

14. SUBJECT TERMS	15. NUMBER OF PAGES
Performance Measures, Computer Measure, Application-Level Fault-Tolerance Interconnection Networks	220
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

Table of Contents

1. Project Goals

The goals of this project were the development of performance measures suitable for use in real-time embedded systems. In particular, we set out to develop a network measure to guide the designer in choosing an appropriate network topology. This measure was meant to combine graph-theoretic concepts in evaluating the underlying reliability of the network, and other means to evaluate the ability of the network to support interprocessor traffic.

The second measure, called the *computer measure*, is meant to evaluate the computer system, in terms that are of importance for real-time applications. It measures the probability of the system being able to deliver, in a timely fashion, a certain amount of critical workload over a given period of operation.

It is possible to integrate both computer and network measures together to form an *integrated* measure of the combined processor and network ability to function appropriately.

2. Project Accomplishments

Our results over this project are summarized below. We intend this write-up to be a brief overview, suitable for quick reading. Fuller details of our work can be found in the papers that are included as an appendix to this report, and in our web page: `http://www.ecs.umass.edu/ece/realtime`.

2.1 Measures

The computer measure that we discovered to be the most useful was the surge-handling ability of processors. Surges can be caused by two types of events: internal and external. Internally, surges can be created as a result of processor failure. When this happens, the critical workload assigned to that processor needs to be distributed among the functioning processors in the system, and arrives as a surge to them. Externally, surges are caused by some unplanned event in the environment, which causes some tasks to be invoked in response.

The key measures of surge-handling ability are:

1. $s(w)$, the deadline by which a processor is able to process a surge of quantum w while still meeting all the deadlines of its own preassigned workload.
2. $t(w)$, the time by which the impact on the processor's schedule of a surge of quantum w decays to zero. Beyond this point, the schedule will be exactly as if the surge had never happened.

We have studied the impact of preemption cost on the surge-handling measure for both the most popular real-time scheduling algorithms: Rate Monotonic and Earliest Deadline

First [9]. We also developed a *sentimental scheduling* algorithm which is useful in lowering preemption costs. This algorithm prevents a higher-priority task from preempting a lower-priority task when the latter is within a short time of finishing (if this can be done without any deadlines being missed). This tends to reduce the number of preemptions, which has an impact on the overhead as well as on the amount of energy consumed.

Several network measures were studied. Among the most useful were the following.

1. Diameter stability. This measure evaluates the expected diameter of a topology, given the probability of failure of the nodes and/or the links. The more stable the diameter, the more impervious the communication costs can be expected to be to component failure.

2. Average distance between node pairs. This measures how badly a given node and link failure probability affects the distance between node pairs. It is a measure of how interprocessor communication costs can be expected to rise as a function of component failure.

3. Largest component size upon disconnection. When enough nodes or links fail that the topology is no longer a connected graph, we are left with a number of disconnected sub-systems. Clearly, it is more convenient for a system, following such disconnection, to have one large (connected) component and several very small ones than to have a large number of very small components. This measure evaluates the expected size of the largest connected component of the network topology, given the probability of node and link failures.

We evaluated, by experiment, the above measures in a variety of network topologies, such as the mesh, toroidal mesh, Mobius graph, chordal ring, and hypercube. Correlated failures were also modeled and the reliability of network topologies in the face of correlated failures were obtained [10].

2.2 Simulator

We built a simulator testbed called TRIDENT to evaluate the network and surge measures. This testbed has a user-friendly Graphical User Interface (GUI) which allows the user to draw the system topology and enter relevant parameters (such as the node and link failure probabilities, recovery time from failure, interprocessor communication loads, task-to-processor allocation scheme, surge-to-processor allocation rules, etc.). It then computes the network and computer measures.

We have also continued work on the RAPIDS simulator, which is meant to evaluate the deadline-meeting performance of real-time workloads under a variety of failure-recovery strategies. RAPIDS has been enhanced to provide the capability of execution-driven simulation [8, 4, 1]. We have also studied the use of importance sampling to speed up the

simulation of rare events. Importance sampling has been studied for some time by the performance-evaluation community, but is generally regarded as a temperamental approach to accelerate rare-event simulations: sometimes, it gives quite inaccurate results. We have demonstrated, by using case studies, ranges of reliability values for which importance sampling provides high accuracy while greatly reducing simulation time. This study provides useful guidance to those seeking to use this scheme [2].

2.3 Application-Level Fault-Tolerance (ALFT)

Application-level fault tolerance (ALFT) is a mechanism by which to reduce the size of the surge that results when a processor suffers transient failure. The idea is to use semantic information from the application to selectively execute only the most important portion, providing somewhat degraded but still acceptable output.

Our work has shown that application-level information can be exploited to greatly reduce the amount of redundancy required to deal with transient failures, which are by far the most common type of failure. For example, in a radar target-tracking application, our approach, required only 15% redundancy to provide complete fault-tolerance against transient faults. Another use of ALFT is in providing a temporary patch in the event of a permanent processor failure, allowing the system more time to execute a recovery algorithm.

ALFT is orthogonal to other approaches to fault-tolerance, so that it can be used either by itself or in combination with them. For example, a designer might use ALFT to guard against transients, and make a small amount of hardware redundancy available, in the form of line-replaceable spares, to deal with permanent failures.

In our work we have established that ALFT has considerable potential to provide relatively inexpensive and effective fault-tolerance. Our results with several data-parallel applications are very promising, achieving in some cases 100% fault tolerance at a cost of 30% or less redundancy [7, 3].

2.4 Synthesis of Interconnection Networks

Our work on developing suitable measures to evaluate interconnection networks led to considering techniques by which to synthesize interconnection networks having desired properties. We implemented a new procedure for automatically synthesizing such networks, which has been surprisingly effective. The technique consists of randomly generating graphs of the desired size and degree (i.e., number of links per node) and then passing them through a set of threshold filters. Each filter removes graphs which fall below a certain threshold with respect to a network measure specified by the user. For example, the user may specify diameter stability, embeddability, scalability, etc. We have shown that such networks are superior to most of the popular networks in use today (e.g., hypercube, mesh, and chordal ring) [5, 6].

3. Technology Transfer

Much of the technology developed in this project is being transferred to the Jet Propulsion Laboratory (JPL). Indeed, JPL has awarded us contracts to modify the RAPIDS testbed to the needs of their REE program, and to evaluate the usefulness of ALFT in their space applications.

4. Educational Contributions

Several graduate students gained exposure to distributed embedded systems as a result of working on this project. Two students completed their MS degrees, with two PhD and one MS student still in process. The graduated students are now working in Lucent Bell Laboratories and Lincoln Laboratory, respectively.

Enclosures

The following are attached as appendices to this report.

1. Papers connected with this research.
2. Master's theses of students supported by this project: Joshua Haines and Gopinath Durairaj.

References

[1] M. Allalouf, J. Chang, G. Durairaj, J. Haines, V.R. Lakamraju, K. Toutireddy, O.S. Unsal, K. Yu, I. Koren and C.M. Krishna, "The RAPIDS Simulator: A Testbed for Evaluating Scheduling. Allocation, and Fault-Recovery in Distributed Real-Time Systems," *Dependable Network Computing*, D. Avresky (Editor), pp. 413-431, Kluwer Academic Publishers, MA, 2000.

[2] G. Durairaj, I. Koren and C.M. Krishna, "Importance Sampling to Evaluate Real-Time System Reliability: A Case Study," to appear, *Simulation,* 2001.

[3] J. Haines, V.R. Lakamraju, I. Koren and C.M. Krishna, "Application-Level Fault Tolerance as a Complement to System-Level Fault Tolerance," *The Journal of Supercomputing*, Special Issue on "Embedded Fault-Tolerant Computing Systems," Vol. 16, pp. 53-68, Kluwer Academic Publishers, MA, 2000.

[4] K. Yu, K. Toutireddy, I. Koren and C.M. Krishna, "Introduction to a Fault-Tolerant Distributed Real-Time System Simulator," *Intern. Journal of Modeling and Simulation,* Vol. 19, No. 1, pp. 7-10, 1999.

[5] V. Lakamraju, I. Koren and C.M. Krishna, "Synthesis of Interconnection Networks: A Novel Approach," *Proc. of the 2000 International Conference on Dependable Systems and Networks,* pp. 501-509, June 2000.

[6] V.R. Lakamraju, I. Koren and C.M. Krishna, "A Randomized Approach to the Synthesis of Interconnection Networks," *Proc. of HPEC'99, Annual Workshop on High Performance Embedded Computing,* Lincoln Lab, pp. 43-44, Sept. 1999.

[7] J. Haines, V.R. Lakamraju, I. Koren and C.M. Krishna, "Development of Application-Level Fault Tolerance in a Real-Time Benchmark," *Proc. of EFTS'98, IEEE Workshop On Embedded Fault-Tolerant Systems,* Boston, May 1998.

[8] M. Allalouf, J. Chang, G. Durairaj, V.R. Lakamraju, O.S. Unsal, I. Koren and C.M. Krishna, "RAPIDS: A Simulator Testbed for Fault- Tolerant Real-Time Systems," *Proc. of HPC'98, Grand Challenges in Computer Simulation,* pp. 191-196, Boston, April 1998.

[9] Z. Koren, I. Koren and C.M. Krishna, "Surge Handling as a Measure of Real-Time System Dependability," *Proc. of the IPPS/SPDP'98 workshop, on Parallel and Distributed Real-Time Systems,* J. Rolim (Ed.), Lecture Notes in Computer Science 1388, Springer 1998, pp. 1106-1116.

[10] V. Lakamraju, Z. Koren, I. Koren and C.M. Krishna, "Measuring the Vulnerability of Interconnection Networks in Embedded Systems," *Proc. of the IPPS/SPDP'98 workshop, on Embedded HPC Systems and Applications,* J. Rolim (Ed.), Lecture Notes in Computer Science 1388, Springer 1998, pp. 919-924.

Appendix 1

Measuring the Vulnerability of Interconnection Networks in Embedded Systems

V. Lakamraju, Z. Koren, I. Koren, and C. M. Krishna

Department of Electrical and Computer Engineering
University of Massachusetts, Amherst, MA 01003

Abstract. Studies of the fault-tolerance of graphs have tended to largely concentrate on classical graph connectivity. This measure is very basic, and conveys very little information for designers to use in selecting a suitable topology for the interconnection network in embedded systems. In this paper, we study the vulnerability of interconnection networks to the failure of individual links, using a set of four measures which, taken together, provide a much fuller characterization of the network. Moreover, while traditional studies typically limit themselves to uncorrelated link failures, our model deals with both uncorrelated and correlated failure modes. This is of practical significance, since quite often, failures in networks are correlated due to physical considerations.

1 Introduction

The interconnection network is an integral part of most embedded systems. It has often as considerable an impact on the system's performance as the nodes themselves. The choice of an appropriate interconnection network is therefore key to determining the performance of the embedded system. Performance measures for interconnection networks are essential to guide the designer in choosing an appropriate topology. In large systems – especially those which must operate for long durations without any possibility of repair – the probability is significant that one or more nodes and/or links are down at any time and this can affect the performance of the system considerably.

Studies of the fault-tolerance of networks have tended to largely concentrate on measures such as classical node (link) connectivity. They measure the extent to which the network can withstand the failure of individual links and nodes while still remaining functional. Such measures are very basic and limited in what they can express of reliability (see [4] for a survey of measures of network vulnerability). They are worst-case measures and convey very little information for designers to use in selecting a suitable topology for the interconnection network in embedded systems.

In this paper we study the vulnerability of an interconnection network to the failure of individual links, using a set of four measures which, taken together, provide a much fuller characterization of the network. Moreover, while traditional studies typically limit themselves to independent link failures, our studies deal with correlated failure modes, as well.

We start in Section 2 by defining four measures of network vulnerability. We follow this in Section 3 with some numerical results. A brief discussion in Section 4 concludes the paper.

2 The Performance Measures

The four performance metrics used to assess network vulnerability can be grouped into two pairs. The first pair assesses the tendency of the topology under study to become disconnected due to link failures. The two measures under this category are:

- 1. The probability that the network becomes disconnected, π_d.
- 2. The size of the biggest connected component, χ_{max}.

The probability that the network becomes disconnected gives us guidance as to the chance that all the processors remain usable (assuming the processors themselves do not fail) by being reachable from every other processor. If the network does get disconnected, we are interested in what happens to the splinters that are left. In particular, we are concerned with whether the graph breaks up into a large number of small components, or whether there is one large component which contains most of the nodes. The latter is obviously preferable. All other things being equal, therefore, we would prefer a network which would disconnect in such a way that the biggest component left after disconnection contains a large fraction of the nodes.

The second pair of measures focuses on node-pair distances. They are:

- 3. The diameter of the network, Δ.
- 4. The average distance between node pairs, $\overline{D}$.

Node pair distances play a role in determining the time it takes for messages to be sent from one node to another. A graph whose diameter is relatively stable is obviously superior to another whose diameter exhibits rapid variations upon link failure.

The notion of diameter stability is not new: the previously-defined measure of edge *persistence* [3] is the minimum number of edges that must be removed to increase the graph diameter. Persistence, however, being a worst-case measure, conveys much less information about graph vulnerability than does the diameter, as a function of the component failure probability.

Inter-node distances play a large role in determining the communication delays between nodes. Algorithms that assign tasks to nodes (processors) have to account for inter-node communication delays when dealing with tasks which communicate with one another. The smaller the delays between the nodes, the greater are the options available to the task assignment algorithm. This is especially true when the original task assignment (on a computer without any failures) is sought to be done in such a way that any task reassignment required upon failure is reduced. For hard real-time systems, it becomes important that the system state on the failed node be transferred to another node with very little delay. This parameter gives a good estimate of the amount of delay that would be involved in the movement of data that would be required to re-establish the state. A close estimation of such delays can help in the efficient calculation of fault-recovery policies[2]. It also gives an indication of how closely the nodes are connected to each other and this can help in the scheduling of tasks.

3 Simulation Models and Results

We consider two link failure models: uniform and clustered. In the uniform model, link failures follow an IID (independent and identically distributed) stochastic process. Each link fails with probability p_f, and link failures are independent of one another. In the clustered model, a probability of either $p - \delta$ or $p + \delta$ (for some given p, δ) is randomly selected for each *node*. Each link incident on a node fails with the failure probability drawn for that node. This failure mechanism results in adjacent links being correlated

with regard to faults, and consequently, in bigger clusters of faulty links and of fault-free links than those generated by the IID link failures. δ is the *clustering parameter*. The greater the value of δ, the more clustered the failing links will be. Note that since the failure probability is applied twice to the same link, the actual probability of a random link failure in the correlated model is $p_f = 1 - (1-p)^2$.

Three different classes of topologies have been used for the simulation runs, namely, the mesh, the hypercube and the generalization of a chordal ring proposed by Arden and Lee[1]. This is a chordal ring in which extra links are added (apart from the 2 links connected to each of its neighbors) among the nodes in some regular fashion. The exact placement of these extra links has an impact on both the traditional measures as well as the ones proposed here.

All the simulation runs were on networks of 64 nodes. Some of the networks had degree 4 and the rest degree 6. A simple rectangular mesh as well as its counterpart, the mesh torus (a mesh with an end-around connection) and both 2-D and 3-D meshes were tested. Simulation runs were performed to measure the effect of the link failure probability, p_f, as well as the the effect of the clustering parameter, δ, on the different performance measures, for the mentioned graph families. A number of interesting results can be concluded from the plots.

Figures 1, 2, 3 and 4 depict the dependence on the link failure probability p_f (in the IID link failure model) of the probability of network disconnection π_d, the maximum component size χ_{max}, the diameter Δ, and the average node-pair distance $\overline{D}$, respectively, for the different topologies.

The conclusions we can derive from these figures are as follows. Though the rectangular mesh is the topology of choice when scalability is concerned, it is certainly not the best topology when considering resistance to link failures. The probability that the network becomes disconnected increases rapidly as the probability of link failure increases. The other topologies in its class do better in all the other parameters as well. Similarly, among the degree-6 networks, the 3-D mesh performs very badly compared to the other topologies in its class.

The chordal ring of degree 4 has better diameter stability compared to the mesh torus. One word of caution though: The diameter of the chordal ring depends on the placement of the extra links (i.e. not those connected to immediate neighbors). For the simulations, an extensive search was performed to find a placement of links which would result in the minimum diameter.

The chordal ring of degree 6 performs only marginally better than the hypercube and the 3-D mesh torus in the diameter and average distance measures.

Figures 5 and 6 show the dependence of the probability of network disconnection, π_d, and that of the maximum component size, χ_{max}, respectively, on the fault clustering parameter, δ, for several graph topologies. The incidence of disconnected graphs increases with the failure clustering (even though the link failure probability remains the same). Again, the meshes without the end-around connection perform badly compared to the other networks. Each family of graphs has a distinctive sensitivity to the level of failure clustering.

The size of the largest connected component decreases as the degree of clustering increases. Also, the maximum component size is dependent on the clustering of links in the topology. This is illustrated in Figure 6 with the two types of the chordal ring. The *good placement* refers to an optimal placement of the links whereas *bad placement* refers to a sub-optimal placement. The dependence of the extra link placement on the component size becomes negligible as the degree of the network increases.

Our experiments also showed that the two other measures, namely, the diameter and the

average node-pair distance (in graphs that remain connected) are not very sensitive to the failure clustering, with the diameter being slightly more sensitive than the average distance. This result holds across graph families.

4 Discussion

In this paper, we have studied the vulnerability of various topologies to link failure. These results – and others like them – can be used by designers in choosing the appropriate topology. We have confined ourselves to a set of symmetric networks: we plan to extend our studies to irregular topologies.
There is also room for modeling correlated failures in other ways. One of them would be to use a "wave-propagation" model, in which the effect of the correlated failure at a node ripples through the network so that all the links which are at the same distance from the failed node has the same probability of failure and this probability decreases as the distance increases. It would also be interesting to look at the combined effect of both node and link failures.

Acknowledgment
This effort was supported in part by the Defense Advanced Research Projects Agency and the Air Force Research Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-96-1-0341, order E349. The government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon.
The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, or the Defense Advanced Projects Agency, Air Force Research Laboratory, or the U. S. Government.

References

1. B.W.Arden and H.Lee. "Analysis of Chordal Ring Networks", *IEEE Transactions on Computers* C-30, 1981.
2. M.Berg and I. Koren, "On Switching Policies for Modular Fault-Tolerant Computing Systems", *IEEE Transactions on Computers* Vol. C-36, 1987.
3. F. T. Boesch, F. Harary, J. A. Kabell, "Graphs as Models of Communication Network Vulnerability: Connectivity and Persistence," *Networks*, Vol. 11, 1981.
4. M.Choi and C.M.Krishna, "On Measures of Vulnerability of Interconnection Networks", *Microelectronics and Reliability* Vol 29, No. 6, 1989.
5. T.Cormen, C. Leiserson and R.Rivest, *Introduction to Algorithms*, Cambridge: MIT Press, 1990.
6. C. M. Krishna and K. G. Shin, *Real-Time Systems*, New York: McGraw-Hill, 1997.

This article was processed using the LaTeX macro package with LLNCS style

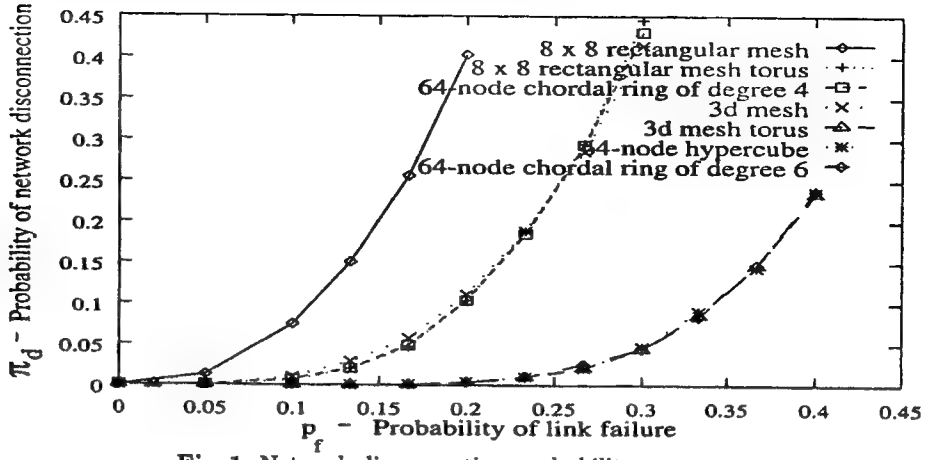


Fig. 1. Network disconnection probability versus p_f

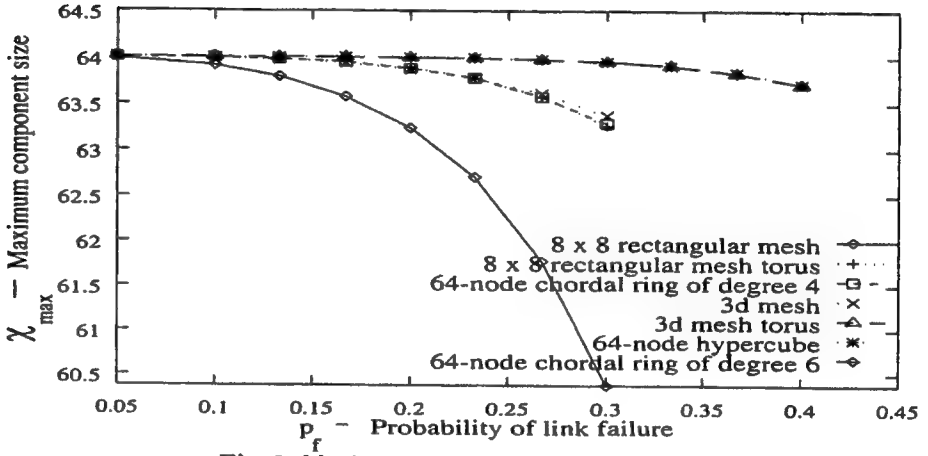


Fig. 2. Maximum component size versus p_f

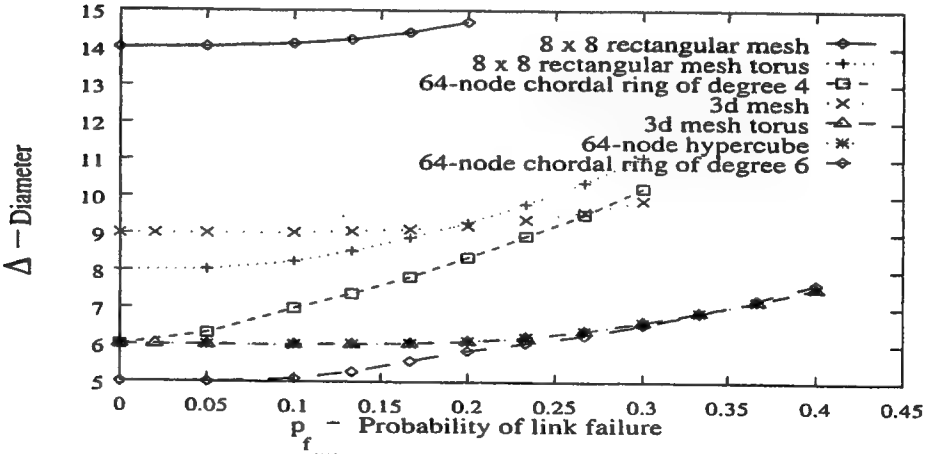


Fig. 3. Diameter versus p_f

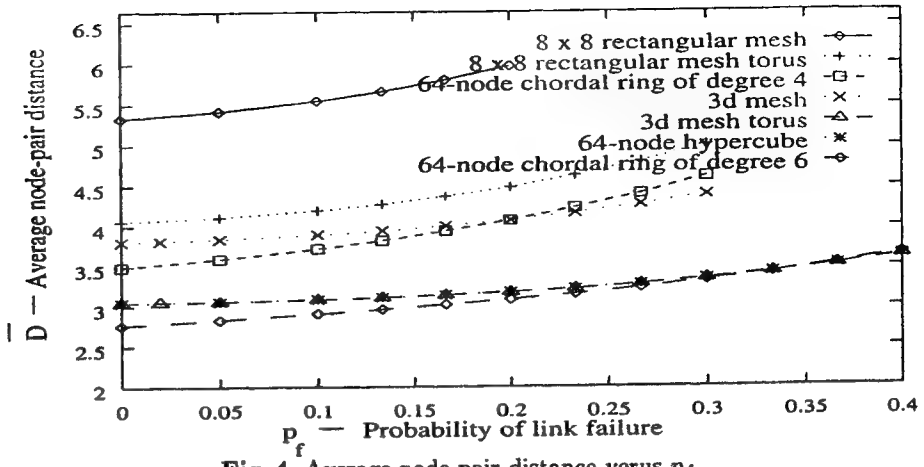


Fig. 4. Average node-pair distance versus p_f

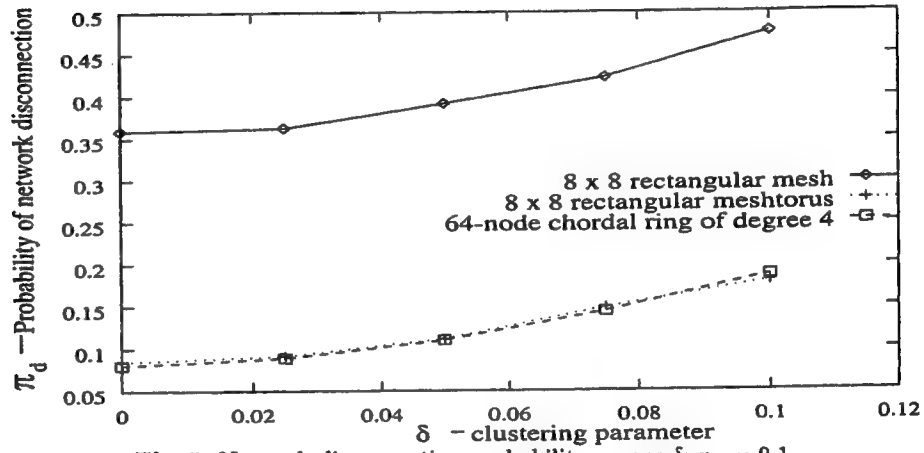


Fig. 5. Network disconnection probability versus δ; $p_f = 0.1$

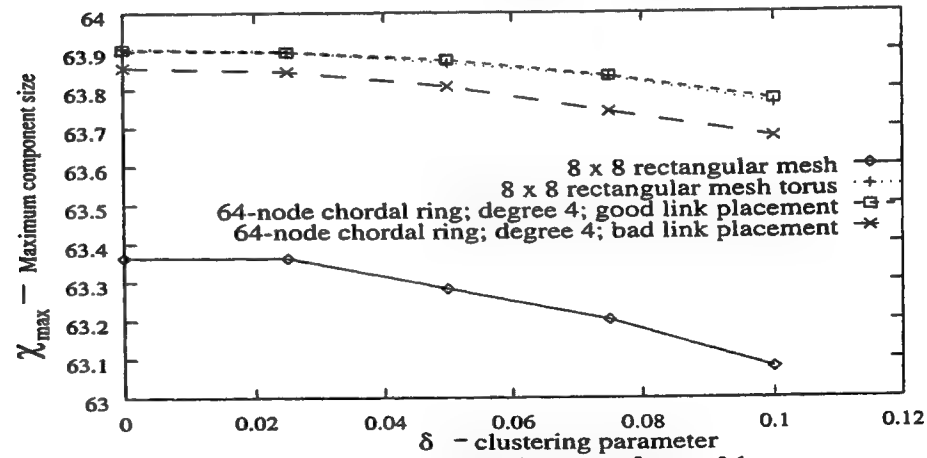


Fig. 6. Maximum component size versus δ; $p_f = 0.1$

- A technique such as this is applicable in a wide variety of distributed real-time applications particularly those exhibiting data parallelism. Examples of these include target tracking applications and compilation of data for automated navigation.
- The extent to which the secondary part mimics the primary depends on the contribution of application-level fault tolerance required to achieve the required amount of reliability.
- Incorporating both the primary and secondary parts within each task of an application, rather than running two, independent copies of the application, allows for many more options for fault tolerance, while simultaneously reducing the amount of system resources required to achieve a given level of reliability.

An existing real-time system simulator, the RAPIDS simulator provides the researcher with a test bed, with which to develop and test a wide variety of system-level fault tolerance techniques. Incorporated with this system simulator is a benchmark application. Thus the system designer is able to experiment with both system and application-level fault detection and fault recovery techniques. The application considered is the Honeywell Real-Time Multi-Hypothesis Tracking (RTHT) Benchmark, and is a general purpose, parallel, target tracking benchmark. With these tools, it is shown that the techniques discussed yield a considerable improvement over the benchmark as it was originally written. This improvement is measured by comparing the total percentage of targets which are tracked as well as with a broader measure involving the total of the likelihoods of all hypotheses in the system, for both the original and the fault tolerant versions. Data has been collected from runs of the benchmark both in conjunction with the simulator and in a stand-alone form. The full paper will contain plots that reflect the dependency of:

- The redundancy and the amount of fault tolerance
- The system resource requirements and the redundancy

References

A.L. Liestman and R. H. Campbell. A Fault-Tolerant Scheduling Problem, *IEEE Transactions on Software Engineering,* November 1986.

B. VanVoorst; R. Jha; L. Pires; M. Muhammad. Implementation and Results of Hypothesis Testing from the C^3I Parallel Benchmark Suite, *Proceedings of the 11th International Parallel Processing Symposium,* 1997

M. Allalouf; J. Chang; G. Durairaj, V.R. Lakamraju; O.S. Unsal; I. Koren; C.M. Krishna. RAPIDS: A Simulator Testbed for Distributed Real-Time Systems, *ASTC* 1998.

, , 1–16 ()

Appendix 3

Application-Level Fault Tolerance as a Complement to System-Level Fault Tolerance

JOSHUA HAINES jhaines@ecs.umass.edu
VIJAY LAKAMRAJU vlakamra@ecs.umass.edu
ISRAEL KOREN koren@ecs.umass.edu
C. MANI KRISHNA krishna@ecs.umass.edu
Electrical and Computer Engineering Dept., University of Massachusetts, Amherst, MA 01003

Abstract. As multiprocessor systems become more complex, their reliability will need to increase as well. In this paper we propose a novel technique which is applicable to a wide variety of distributed real-time systems, especially those exhibiting data parallelism. System-level fault tolerance involves reliability techniques incorporated within the system hardware and software whereas application-level fault tolerance involves reliability techniques incorporated within the application software. We assert that, for high reliability, a combination of system-level fault tolerance and application-level fault tolerance works best. In many systems, application-level fault tolerance can be used to bridge the gap when system-level fault tolerance alone does not provide the required reliability. We exemplify this with the RTHT target tracking benchmark and the ABF beamforming benchmark.

Keywords: distributed real-time systems, fault tolerance, checkpointing, imprecise computation, target tracking, beam forming.

1. Introduction

In a large distributed real-time system, there is a high likelihood that at any given time, some part of the system will exhibit faulty behavior. The ability to tolerate this behavior must be an integral part of a real-time system. Associated with every real-time application task is a deadline by which all calculations must be completed. In order to ensure that deadlines are met, even in the presence of failures, fault tolerance must be employed. In this paper we consider fault tolerance at two separate levels, system-level and application-level.

System-Level Fault Tolerance encompasses redundancy and recovery actions within the system hardware and software. While system hardware includes the computing elements and I/O (network) sub-system, the system software includes the operating system and components such as the scheduling and allocation algorithms, checkpointing, fault detection and recovery algorithms. For example, in the event of a failed processing unit, the component of the system responsible for fault tolerance would take care of rescheduling the task(s) which had been executing on the faulty node, and restarting them on a good node from the previous checkpoint.

Application-Level Fault Tolerance encompasses redundancy and recovery actions within the application software. Here various tasks of the application may communicate in order to learn of faults and then provide recovery services, making use of some data-redundancy. In certain situations, we find that fault tolerance at the

Appendix 2

Development of Application-Level Fault Tolerance in a Real-Time Benchmark

J. Haines, V. Lakamraju, I. Koren and C. M. Krishna
Department of Electrical and Computer Engineering
University of Massachusetts
Amherst, MA 01003

Paper Abstract

Multiprocessor systems are in real need of fault tolerance features if dependable service is to be expected. As these systems become more complex, reliability will need to increase as well. System-level fault tolerance alone does not always suffice to give the required amount of reliability in distributed real-time systems. In this paper, we assert that for high reliability and accuracy of results, a combination of system-level fault tolerance and application-level fault tolerance works best. In most systems, fault tolerance is achieved through the process of fault detection and location, fault containment and fault recovery. The process of fault recovery can take a relatively long time when done totally at the system-level which might not be acceptable in real-time systems. In order to ensure that deadlines are met, application-level fault tolerance can be employed in many real-time applications.

Some important points which we stress regarding the integration of application and system-level fault tolerance:

- A combination of system-level fault tolerance and application-level fault tolerance can yield much higher reliability, than either one alone.
- A general, system-level fault tolerance method may involve actions such as the moving of checkpoints, reconfiguration and restarting of tasks on non-faulty nodes. These actions may take a non-negligible amount of time to complete, from the point of view of the application. We believe that application-level fault tolerance provides a method by which the application can fill the gap between when a fault occurs and when the system-level recovery is complete, in order to best meet its deadlines. This can substantially decrease the overall fault recovery time, thereby decreasing overall system vulnerability. Decreasing the stress on the speed of a system-level recovery technique might also allow the system more recovery options to choose from for a given circumstance.
- In the technique which we discuss, application-level redundancy takes the form of a primary part and a lower priority secondary part within each application task. The primary part is responsible for the computation that is required of it as part of the original application and the secondary part provides the necessary fault tolerance by running part of its neighbor's primary work. This allows tasks to make better use of processor time and system resources which might have gone unused, due to the periodic nature of real-time task sets.

application-level can greatly augment the overall fault-tolerance of the system. For example, if a task's checkpoint is very large, application-level fault tolerance can help mask a fault while the system is moving the large checkpoint and restarting the task on another node.

N-Modular Redundancy is a well-known fault tolerance technique. A number of identical copies of the software are run on separate machines, the output from all of them is compared, and the majority decision is used [1]. This technique however, involves a large amount of redundancy and is thus costly.

The recovery block approach combines elements of checkpointing and backup alternatives to provide recovery from hard failures [2]. All tasks are replicated but only a single copy of each task is active at any time. If a computer hosting an active copy of a task fails, the backup is executed. The task may be completely restarted (which increases the chances of a deadline miss) or else executed from its most recent checkpoint [4]. The later option requires that the active copy of the task periodically copy (checkpoint) its state to its backups. This can entail a large amount of overhead, especially when the state information to be transferred is large. Such is the case with the applications that we are dealing with.

Another common technique is the use of less precise (i.e., approximate) results [3], obtained by operating on a much smaller data set, using the same algorithm. A data set can be chosen such that a sufficiently accurate result can be obtained with a greatly reduced execution time. A smaller data set is chosen either by prioritizing the data set or by reducing the granularity. Examples of such applications are target tracking and image processing, where it is better to have less precise results on time, rather than precise results too late or not at all. Our recovery technique caters to applications that exhibit data- parallelism, involves a large data set and can make do with a less precise result for a short period of time.

Our approach makes use of facets of the recovery block technique and employs reduced precision state information and results in order to tolerate faults. We employ a certain degree of redundancy within each of the parallel processes. The application as a whole is able to make use of that redundancy in the event of a fault to ensure that the required level of reliability is achieved. We consider only failures that render a process' results erroneous or inaccessible. In the case of such a fault, the redundant element's less precise results are used instead of those from the failed process. In this way, our technique can provide a high degree of reliability with only a small computational overhead in certain applications.

Section 2 introduces the RTHT and ABF benchmarks that will be used to demonstrate our technique. In Section 3 we describe in detail our application-level fault tolerance technique. Section 4 analyzes the effectiveness of this technique when used in conjunction with each of the benchmarks, and Section 5 concludes the paper.

2. The Benchmarks

Each of the benchmarks has the form shown in Figure 1. There are multiple, parallel *application* processes, which are fed with input data from a source - in this case, a *source* process which simulates a radar system or an array of sonar sensors. When

the parallel computations are complete, the results are output to a *sink* process, simulating system display or actuators. Our technique is concerned with the ability to withstand faults at the parallel processes.

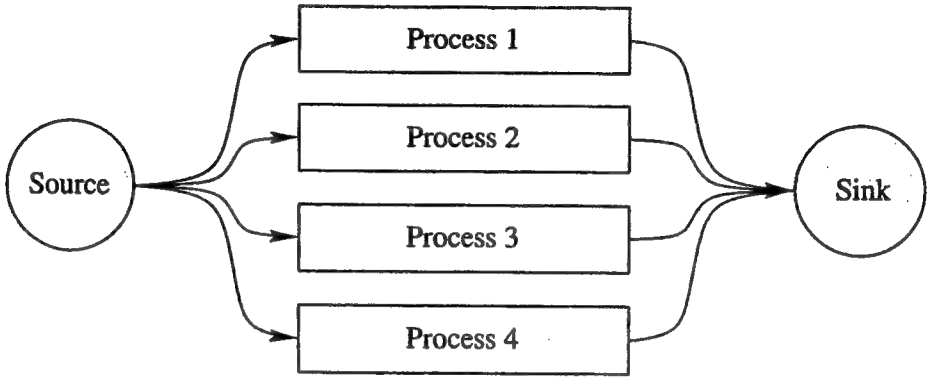


Figure 1. Software architecture of both the RTHT and ABF benchmarks.

2.1. The RTHT Target Tracking Benchmark

The Honeywell Real-Time Multi-Hypothesis Tracking (RTHT) Benchmark [6, 7], is a general-purpose, parallel, target-tracking benchmark. The purpose of this benchmark is to track a number of objects moving about in a two-dimensional coordinate plane, using data from a radar system. The data is noisy, consisting of false targets and clutter, along with the real targets. The original, non-fault-tolerant application consists of two or more processes running in parallel, each working on a distinct subset of the data from the radar. Periodically, frames of data arrive from the radar, or source process in this case, and are split among the processes for computation of hypotheses. Each possible track has an associated hypothesis which includes a figure of likelihood, representing how likely it is to be a real track. A history of the data points and a covariance matrix are used in generating up-to-date likelihood values.

For every frame of radar data, each parallel process performs the following steps: 1) Creation of new hypotheses for each new data point it receives, 2) Extension of existing hypotheses, making use of the new radar data and the existing covariance matrix, 3) Participation in system-wide compilation or ranking of hypotheses, led by a *Root* application process, and 4) Merging of its own list of hypotheses with

the system-wide list that resulted from the compilation step. The deadline of one frame's calculations is the arrival of the next frame.

By evaluating the performance of the original, non-fault-tolerant, benchmark when run in conjunction with our RAPIDS real-time system simulator [9], it became apparent that despite the inherent system-level fault tolerance in the simulated system, the benchmark still saw a drastic degradation of tracking accuracy as the result of even a single faulty node. Even if the benchmark task was successfully reassigned to a good node after the fault, the chances that it had already missed a deadline were high. This was in part due to the overheads associated both with moving the large process checkpoint over the network and with restarting such a large process. Once the process had missed the deadline, it was unable to take part in the compilation phase and had to start all over again and begin building its hypotheses anew. This took time, and caused a temporary loss of tracking reliability of up to five frames. Although better than a non-fault-tolerant system, in which that process would simply have been lost, it was still not as reliable as desired.

We decided to address two points, in order to improve the performance of the benchmark in the presence of faults: 1) The overhead involved with moving such a large checkpoint and 2) A source of hypotheses for the process to start with after restart.

Our measure of reliability is the number of *real targets* successfully tracked by the application (within a sufficient degree of accuracy) as a fraction of the exact number of real targets that should have been tracked. To simplify this calculation, the number of targets is kept constant and no targets enter or leave the system during the simulation.

2.2. The ABF Beam Forming Benchmark

The Adaptive Beam Forming (ABF) Benchmark [8] is a simulation of the real-time process by which a submarine sonar system interprets the periodic data received from a linear array of sensors. In particular, the goal is to distinguish signals from noise and to precisely identify the direction from which a signal is arriving, across a specified range of frequencies. In this implementation, the application receives periodic samples of data as if from the linear sensor array. The data is generated so that it contains four reference *beams*, or signals, arriving from distinct locations in a 180-degree field of view, along with random noise.

The application itself consists of several application processes, each attempting to locate beams at a distinct subset of the specified frequency range. Frames of data for each frequency are "scattered" periodically from the source process. Output, in the form of one beam pattern per frequency, is "gathered" by the sink process. Figure 2 depicts a typical beam pattern output, shown here at frame 18, frequency 250Hz, with reference beams at -20, -60, 20 and 60 degrees.

Each application process performs calculations according to the following loop of pseudo-code, for each frame of input.

```
for_each ( frequency ) {
```

```
    Update dynamic weights.
    for_each ( direction of arrival) {
      Search for signal, blocking out interference
      from other directions and frequencies.
    }
}
```

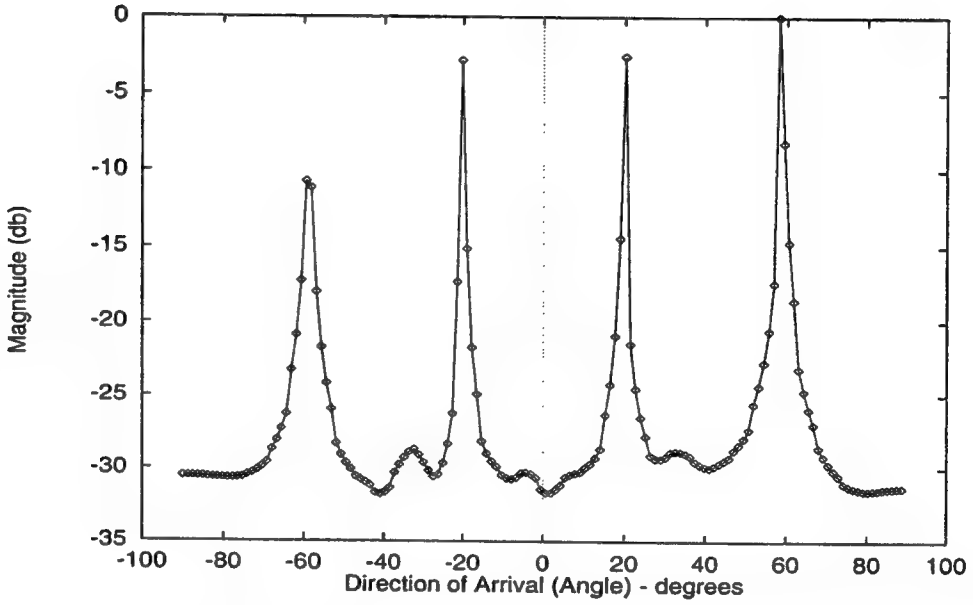


Figure 2. Typical beam pattern output.

For each frequency, the process first updates a set of weights that are dynamically modified from frame to frame. Applying these weights to the input samples has the effect of forming a beam which emphasizes the sound arriving from each direction. The process searches in each possible direction (-90 to 90 degrees) for incoming signals. The granularity of this direction is directly related to the number of sensors.

In addition, at the start of a run, there is an initialization period in which the weights are set to some initial values, and then 15 to 20 frames are necessary to "learn" precisely where the beams are.

It is evident that this sort of application faces reliability problems similar to those of the RTHT benchmark. If a processing element fails, all output for those frequencies is lost during the down time, and when the lost task is finally replaced by the system, it has to go through a startup period all over again. Here, too, the data sets of these processes are very large, creating a considerable overhead if checkpointing is employed. To avoid the delay associated with this overhead, be able to maintain full output during the fault, and provide quick restart after the fault, application-level fault tolerance must be employed.

We evaluate the quality of the ABF output with two tests applied to the resultant beam pattern. In the Placement Test we check whether the direction of arrival of the beam has been detected within a certain tolerance. In the Width Test the aim is to determine how accurately the beam has been detected by measuring the width of the beam, in degrees, at 3db down from the peak. A beam that passes both tests is considered to be correctly detected.

3. Implementation of Application-Level Fault Tolerance

Our technique uses redundancy in the form of extra work done by each process of the application. Each process takes, in addition to its own distinct workload, some portion of its *neighbor*'s workload, as shown in Figure 3. The process then tracks beams or targets for both its own work and overlaps part of its neighbor's, but makes use of the redundant information only in case this neighbor becomes faulty. We now explain briefly how the data set is divided, how the application might learn of faults, and how it would recover from them.

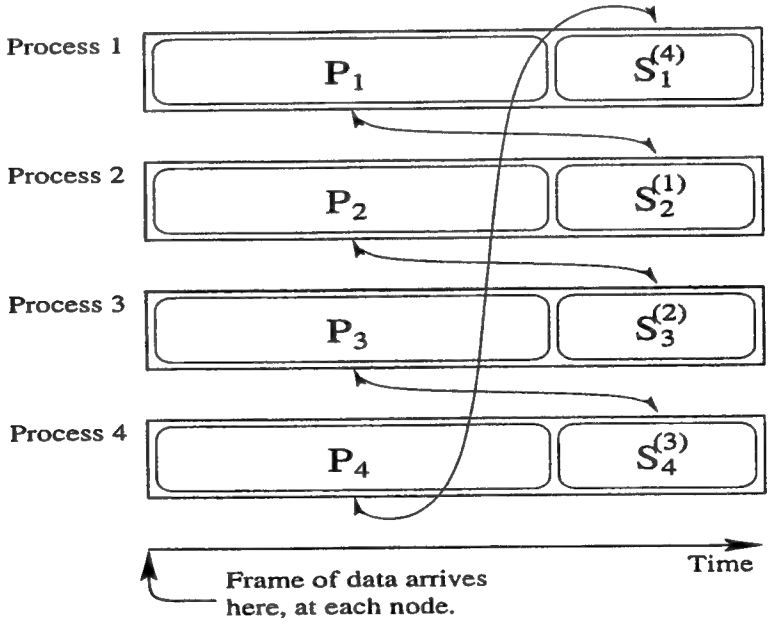


Figure 3. Architecture of both benchmarks with application-level fault tolerance.

3.1. Division of Load

The extent of duplication between two neighboring nodes will greatly affect the level of reliability which can be achieved. Duplication arises from the way we divide the

data set among the parallel processing nodes. First, each frame of data is divided as evenly as possible among the nodes. The section of the process that takes on this set of data is the primary task section, P_i. Then we assign each node, n_i, some additional work: part of its neighbor, n_{i-1}'s, primary task. The section of the process that takes on this set of data is the secondary task section, S_i. In other words,

- The *primary task section*, P_i, refers to the calculations which node n_i carries as part of the original application.
- The *secondary task section*, S_i, refers to the calculations which node n_i carries out as a backup for its neighbor, n_{i-1}. Node n_1 hosts the secondary corresponding to the primary running on the highest numbered node. The secondary section, S_i, will be kept in synchronization with the primary P_{i-1}.

3.2. Detection of Faults

There are two ways in which fault detection information can reach the various application processes. In the first, the system informs the application of a faulty node, and the second is through specific timeouts at the phase of the application where communication is expected. The former would typically incur the cost of periodic polling, while the latter could result in late detection of the fault. Although the exact integration of application-level fault tolerance would vary depending on the fault detection technique chosen, the effectiveness of our technique should not.

3.3. Fault Recovery

If, at a deadline prior to that of the frame, node n_i is discovered to be faulty and is unable to output any results, then node n_{i+1} which is serving as its backup will send as output S_{i+1}'s data in place of the data that n_i is unable to supply. In the meantime, the system will be working on replacing or restarting the process that was interrupted by the fault. In fact, the system's job here is made easier by the fact that if the process has to be restarted on another node, the process data segment no longer needs to be moved. When the process is rescheduled, it will make use of the information maintained by its secondary on its behalf in order to pick up where it left off before the fault. This way, the application fault tolerance is able to work in conjunction with the system fault tolerance. This will help even in the case of transient faults, in that the application-level fault tolerance allows more leeway to postpone the restarting of the process on another node, in the hope that the fault will soon disappear.

3.4. Extension to a higher level of redundancy

Our technique guarantees the required reliability in the presence of one fault but could also withstand two or more simultaneous failures depending on which nodes

are hit by the faults. For example, in a six-node system if the nodes running processes 1, 3, and 5 fail, the technique would still be able to achieve the required reliability. Of course, this is contingent on the assumption that the processes on the faulty nodes are transferred to a safe node and restarted by the beginning of the next frame.

3.5. Benchmark Integration Specifics

We next discuss specific details regarding the application of our technique to each of the benchmarks.

3.5.1. RTHT benchmark In the RTHT Benchmark, the "unit of redundancy" is the hypothesis. That is, each secondary task section creates and extends some fraction of the total number of hypotheses created and extended by the process for which it is secondary. The amount of secondary redundancy is expressed as a percentage of the number of hypotheses extended by the primary.

Redundancy is implemented in the following way: At the beginning of each frame, the source process broadcasts the input radar data, and hypotheses are created and extended as before, with the exception that additionally the secondary extends a percentage of those extended by the corresponding primary. The secondary section S_i is kept in synchronization with primary P_{i-1} via the compilation process, which in this case is again a process-level broadcast communication, so that no extra communication is necessary. If node n_i is discovered to be faulty and is unable to participate in the compilation of that frame, then node n_{i+1} which is serving as its backup will make use of S_{i+1}'s data in the compilation process in place of the data that n_i is unable to supply.

When the process is rescheduled, it will make use of the hypotheses extended by the secondary on its behalf so as to pick up where it left off. This information is obtained from the secondary process by way of compilation - the newly rescheduled process merely listens in on the compilation process and copies those hypotheses which have been extended by its secondary.

3.5.2. ABF benchmark There are two ways in which we have integrated application-level fault tolerance with the ABF Benchmark. They differ in the manner in which the secondary abbreviates the calculations of the primary so as to obtain a full set of results. The methods are:

- The Limited Field of View (Limited FOV) Method in which the secondary looks for beams at every frequency as in the primary, however it searches only a subsection of the primary's field of view (divided into one or more segments). Ideally the secondary will place these "windows" at directions in which beams are known to be arriving. We impose a minimum width of these windows, due to the fact that if an individual window is too narrow, the output could always (perhaps erroneously) pass the width-based quality test, described in section 2.

The amount of redundancy is expressed as the percentage of the field of view searched by the secondary.

- The Reduced Directional Granularity Method in which the secondary looks for beams at every frequency and in every direction, but with a reduced granularity of direction. The amount of redundancy is expressed as a percentage of the original granularity computed by the primary.

Both techniques serve to reduce the computational time of the secondary task set, while maintaining useful system output. In addition, the two techniques may be employed concurrently in order to further reduce the computational time required by the secondary task.

To implement either variation of the technique, the input frame of data is scattered a second time from the source to the application processes. This is time - rotated, so that each process receives the input data of the process for which it is a secondary. Each process first carries out its primary computational tasks, and then carries out its secondary task. At the frame's deadline, if a process is detected to be down, the sink will gather output from the non-faulty processes, including the backup results from the process that is secondary to the one that is faulty. In the event of an application process being restarted after a fault, it will receive the current set of weights from its secondary in order to jump-start its calculations.

Some synchronization between primary and secondary is required in the Limited FOV Method. It is a small, periodic communication in which either the sink process or the primary itself tells the secondary at what frequencies and directions it is detecting beams. Such synchronization is not necessary for the Reduced Granularity Method.

4. Results

4.1. The RTHT Benchmark

When applied to the RTHT benchmark, we found that only a small amount of redundancy between the primary and secondary sections is necessary in order to provide a considerable amount of fault tolerance. Furthermore, the increase in system resource requirements, even after including overheads of the technique's implementation, is minimal compared to that of other techniques, in achieving the same amount of reliability. These points are demonstrated in Figures 4, 5, and 6. Each run contains 30 targets which remain in the system until the end of the simulation (the 30th frame), as well as some number of false alarms. The case when only system-level fault tolerance exists corresponds to the case when the secondary extends 0% of the primary hypotheses.

In Figure 4 we see the number of targets which are successfully tracked, when we have just two application processes and a fault occurs at frame 15. (In this case there were roughly 80 false alarms per frame of data.) In this run, 15% redundancy allows us to track all of the real targets, despite the fault. We can attribute the fact that a small amount of redundancy can have a great effect on the tracking stability,

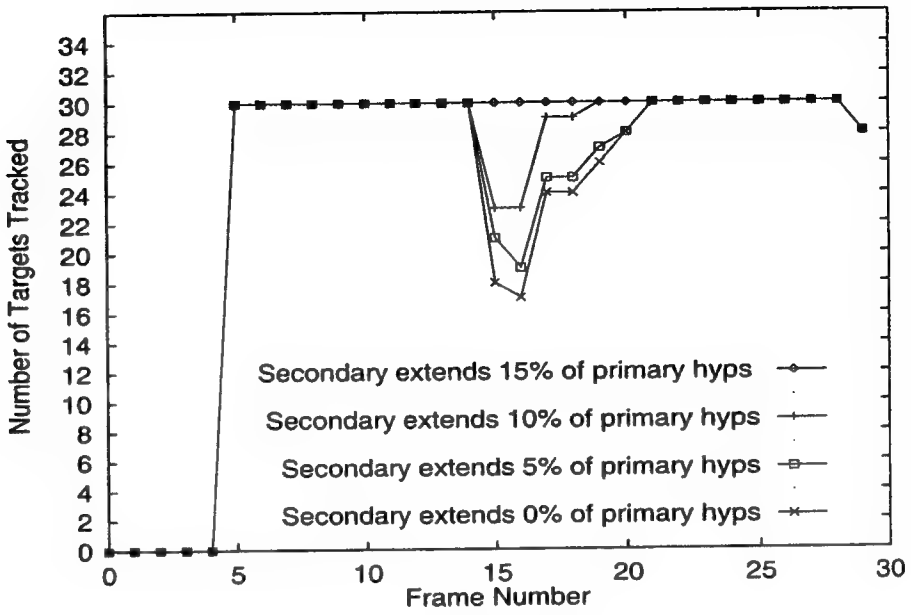


Figure 4. Tracking accuracy, in number of real targets tracked for a given percentage of redundancy.

to the fact that the hypotheses which are being extended by the secondary are the ones *most likely* to be real targets. At the beginning of the compilation phase, each application process sorts its hypotheses, placing the *most likely* at the head of the list for compilation. Thus, at the beginning of the next frame, each application process and its secondary begin extending those hypotheses with the highest chance of being real targets.

To refine this point, Figure 5 shows the average percentage of redundancy required for a given number of application processors and a single fault, as before. The amount required shows a gradual decrease as we add more processors. We can attribute this to the fact that the chance of a single process containing a high percentage of the real targets decreases as processors are added.

In addition, a proportionately small load is imposed on the processor by the computation of the secondary task set, as seen in Figure 6. This can be attributed to the fact that a hypothesis whose position and velocity are known precisely, does not take as much time to extend compared to those hypotheses which are less well-known. And since the *most likely* hypotheses are generally the most well-known *and* are the hypotheses which the secondary extends, the amount of processor time taken to execute the secondary task is proportionally much smaller.

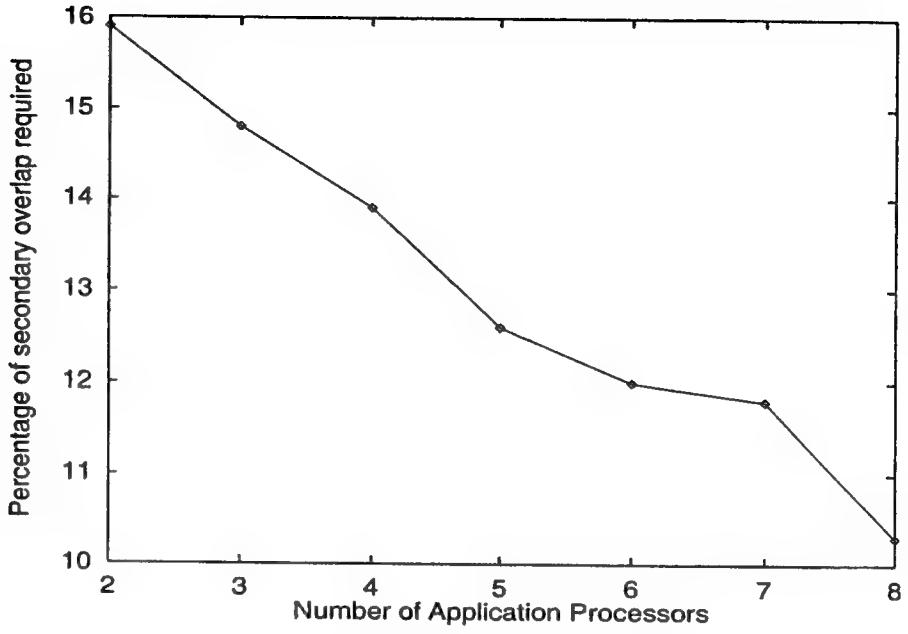


Figure 5. Average minimum percentage of secondary overlap required to miss no targets despite one node being faulty.

4.2. ABF Benchmark Results

When we integrate application-level fault tolerance with the ABF benchmark, we find that only a small amount of redundancy is necessary to ensure complete masking of single frame faults. With either variation (reduced granularity or limited FOV method) we see that a secondary redundancy of 33% is adequate to provide complete and accurate results in the faulty frame and the following frames (after the faulty process is restarted). If we combine the two techniques, we see an even further reduction in the computational effort imposed by the secondary in order to mask the fault. We have not taken additional network overhead and/or latency into account in figures of overhead - they refer solely to computational overhead. Network overhead will depend greatly on the medium used. In particular, a shared medium would allow the secondary to "snoop" on the primary's input and output, eliminating the need for additional communication.

All results were obtained by running simulations with 75 sensors and four reference input beams for 50 frames. There are two application processors, and a fault occurs in one of them at frame 30. Results are presented and discussed for three redundancy methods: the Limited FOV method, the Reduced Granularity method and a Combined method (a combination of the first two). The quality of the results is assessed by totalling the number of beams that were tracked successfully. Here, there are four input beams at each frequency and 32 frequencies – making 128

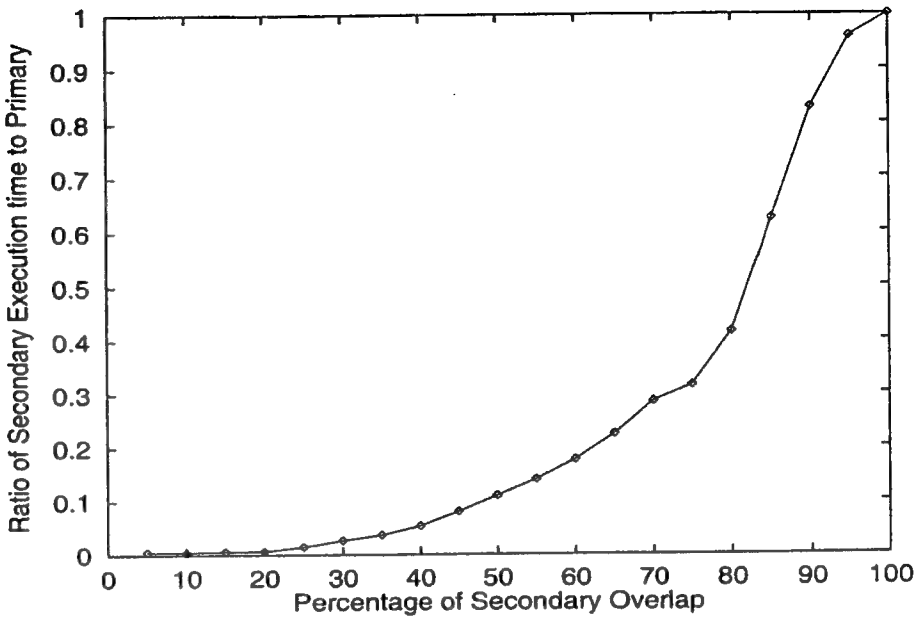


Figure 6. Ratio of time taken to compute the secondary hypothesis to the time to compute the primary hypothesis versus the percentage of secondary overlap.

beams in all. As an example, Figure 7 presents the results for several runs of the ABF benchmark while utilizing the Limited FOV redundancy method alone, with a single processor fault occurring at frame 30 and lasting one frame. We see that a 30% overlap is adequate to preserve all beam information within the system despite the loss of one processor in frame 30. We have tabulated the results for all three methods in Table 1.

4.2.1. ABF Results: Limited FOV Alone As we see in Table 1, roughly 30% secondary overlap is adequate to provide full masking of the fault. The computational overhead imposed by the secondary is about 30%. In addition, Figure 8 shows the rather linear increase in overhead as we increase the fraction of overlap.

Table 1. Amount of secondary overhead imposed by various redundancy methods, each of which is capable of fully masking a single fault.

Redundancy Technique	Secondary Overlap	Computational Overhead
Reduced Granularity	33%	35%
Limited FOV	30%	30%
Combined - 30%FOV,50%Granularity	15%	17%

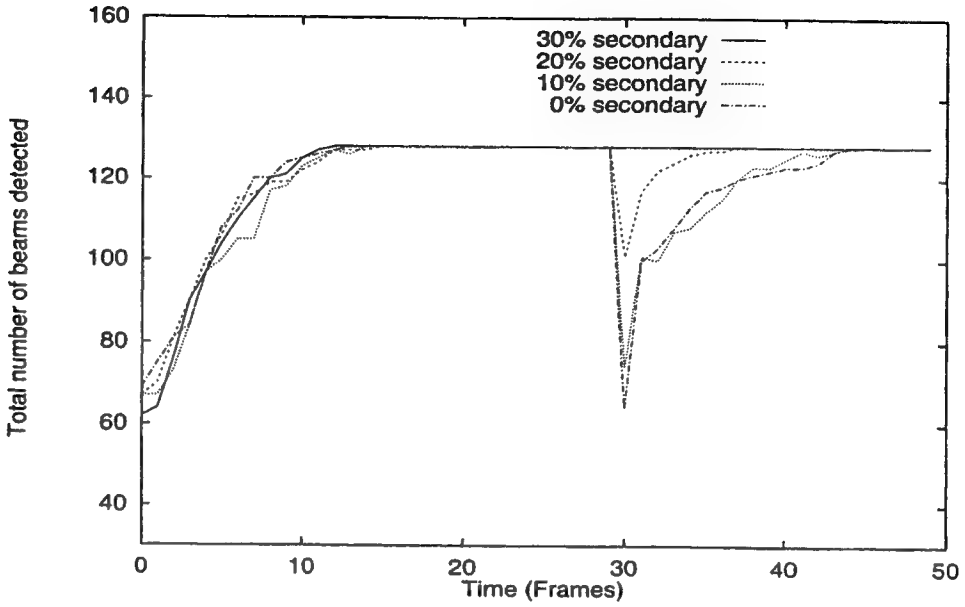


Figure 7. The number of beams correctly tracked in each frame, for the given levels of redundancy, for the Limited Field of View Method. A single process experiences a fault of duration one frame, at frame 30.

Associated with this technique however, is a potential dependence on the number of beams detected in the system, as described earlier. In order to ensure that the width test applied to the output can fail, we impose a minimum window-width. This minimum width dictates that for a given amount of overlap, there is a maximum number of windows in which the secondary may search for beams. If there are more beams than the maximum number of windows then some may be missed by the secondary search, depending on the direction of arrival. However, the system designer can lessen the likelihood of this occurring by carefully choosing the amount of overlap allotted, and tuning the criteria with which areas will be searched by the secondary.

4.2.2. ABF Results: Reduced Granularity Alone Here, too, we see that, according to Table 1, operating the secondary at 33% of the granularity of the primary results in complete masking of the fault, and that this imposes a 35% overhead to the processing node. Figure 8 again shows a linear relationship between the computational overhead and the overlap, and indicates that the overhead of the method itself is a bit higher than that of the Limited FOV method. When considering the Reduced Granularity method, we see no dependence on the number of beams detected, although beams could be missed if their peaks were within a few degrees of each other, and the granularity were very coarse.

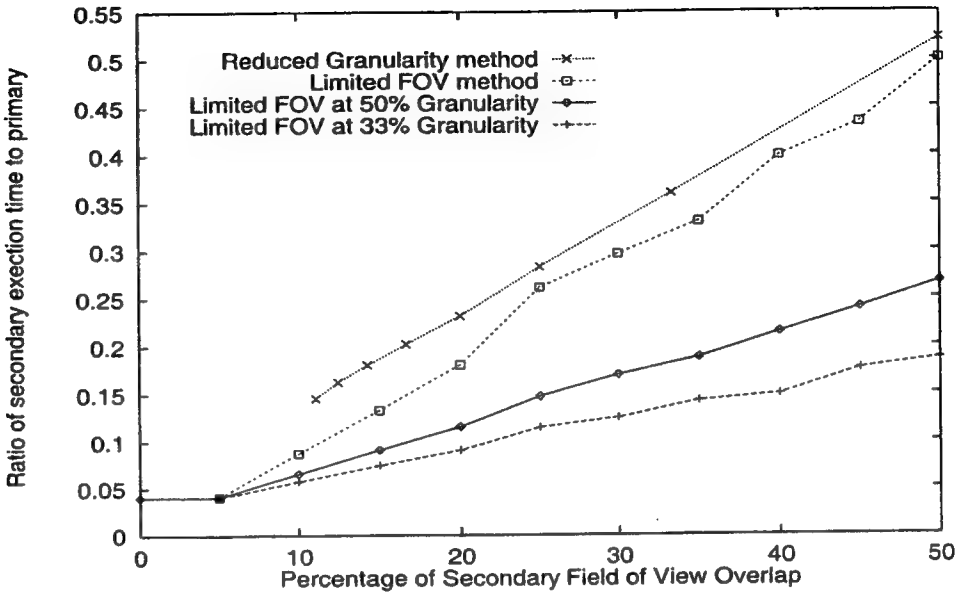


Figure 8. The ratio of secondary to primary execution time for the variations of application-level fault tolerance integrated with the ABF Benchmark versus the percentage of secondary field of view overlap.

4.2.3. ABF Results: Combined methods When we combine these two techniques, we see the greatest reduction in computational overhead of the secondary task. As shown in Table 1, a 30% field of view combined with a 50% granularity maintains the tracking ability similar to that of either one alone, yet cuts the computational overhead nearly in half. This reduction is illustrated in Figure 8, in the lower two curves, representing the overhead imposed as we vary the field of view and make use of 50% and 33% granularity respectively.

5. Conclusions

A high degree of fault tolerance may be obtained with a minimal investment of system resources in applications exhibiting data parallelism, such as the ABF and RTHT Benchmarks. It is achieved through a combination of application-level and system-level fault tolerance. A prioritized ordering within the data set, as in the RTHT benchmark, or a reduced granularity, as in the ABF benchmark, is made use of, to decrease the computational overhead of our technique.

The processes in these benchmarks are very large, so that moving a checkpoint and restarting the task may take a significant amount of time. The application-level fault tolerance is able to ensure that, despite the temporary loss of the task, the required reliability is maintained.

Since the primary and secondary task sets are incorporated within a single application process, the primary is always executed first and the secondary next. Once the primary has completed, it may alert the scheduler, indicating that the secondary need not be executed. It is useful, but not necessary, for the secondary to still be executed, as this allows it to be better synchronized with its primary counterpart. If a fault is detected, the priority of the secondary could be raised, to ensure that it will complete without missing its deadline, and provide the necessary data for compilation.

This technique is a substantial improvement over complete system duplication, in that it does not require 100% system redundancy, but merely adds a small amount of load to the existing system in achieving the same amount of fault tolerance. It differs from the recovery block approach in that the secondary does not have to be cold-started, but is ready for execution when a failure of the primary is detected. In addition, the level of reliability may be varied by varying the amount of redundancy.

In order to integrate such application-level fault tolerance, the designer will need to first determine how to prioritize the data set and/or reduce the granularity in order to define the secondary's dataset. Second, the designer should choose mechanisms by which the secondary gets the input data it needs, is able to output results when necessary, and is able to communicate with the primary for synchronization purposes. Naturally, some sort of fault detection will have to used as well. The designer must carefully weigh the overheads imposed by various methods to achieve fault tolerance and the quality of results that may be obtained from each.

In conclusion, we believe that steps to integrate this technique into the application should be taken right from the early stages of the design in order for this approach to be most effective.

Acknowledgments

This effort was supported in part by the Defense Advanced Research Projects Agency and the Air Force Research Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-96-1-0341, order E349. The government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon.

The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency, Air Force Research Laboratory, or the U. S. Government.

References

1. D.P. Siewiorek and R.S. Swarz *Reliable Computer Systems Design and Evaluation*, 2nd ed. Digital Press, Burlington, MA, 1992.
2. B. Randell. System Structure for Software Fault Tolerance. *IEEE Transactions on Software Engineering*, vol. SE-1, pp. 220-232, 1975.
3. J.W.S. Liu, W. Shih, K. Lin, R. Bettati, and J. Chung. Imprecise Computations. *Proceedings of the IEEE*, vol. 82, No. 1, pp. 83-93, Jan. 1994.

4. N.A. Speirs and P.A. Barrett. Using Passive replicates in Delta-4 to Provide Dependable Distributed Computing. *Proceedings of the Nineteenth International Symposium on Fault-Tolerant Computing*, 1989, pp. 184-190.
5. A.L. Liestman and R.H. Campbell. A Fault-Tolerant Scheduling Problem. *IEEE Transactions on Software Engineering*, vol. SE-12, pp. 1089-1095, Nov. 1986.
6. B. VanVoorst, R. Jha, L. Pires, M. Muhammad. Implementation and Results of Hypothesis Testing from the C^3I Parallel Benchmark Suite. *Proceedings of the 11th International Parallel Processing Symposium*, 1997.
7. D.A. Castanon and R. Jha. Multi-Hypothesis Tracking (Draft). DARPA Real-Time Benchmarks, Technical Information Report (A006), 1997.
8. R. Hamza, Honeywell Technology Center. Sonar Adaptive Beamformer (Draft). DARPA Real-Time Benchmarks, Primary Technical Information Report, 1998.
9. M. Allalouf, J. Chang, G. Durairaj, V.R. Lakamraju, O.S. Unsal, I. Koren, C.M. Krishna. RAPIDS: A Simulator Testbed for Distributed Real-Time Systems. *Advanced Simulation and Technology Conference*, 1998, pp. 191-196.
10. C.M. Krishna and K.G. Shin *Real-Time Systems*, McGraw Hill, New York, NY, 1997.

Appendix 4

Surge Handling as a Measure of Real-Time System Dependability *

Zahava Koren, Israel Koren and C. M. Krishna

Department of Electrical and Computer Engineering
University of Massachusetts, Amherst, MA 01003

Abstract. Traditional reliability measures for computer systems can be classified into *Computer-Centric* or *Application-Centric* categories. The former concentrate on the hardware resources while ignoring the application's needs. The latter focus on the requirements of a specific application which is being executed, thus requiring the knowledge of all the details of the application; information which may not always be readily available. Also, the narrow view on the system's reliability through a single application is too restrictive and provides very limited information regarding the way the system will handle other applications.

In this paper we present new measures for real-time system reliability. These measures are application-*sensitive* rather than application-*centric*, and are especially suitable for systems executing various applications with different attributes, some of which may not be known in advance.

Our proposed measures capture the capability of a real-time system to respond successfully to unexpected surges in the workload. These surges may result from a phase change in the system's mission, an application-related emergency situation or the failure of some system resources. The ability of the system to handle such surges determines, to a large extent, its chances of survival and meeting its applications' deadlines.

1 Introduction

In this paper, we discuss the merits of using the surge-handling capability of a real-time embedded system as a measure of its reliability. Such systems are increasingly used to control life-critical processes such as fly-by-wire aircraft, nuclear reactors, etc. Reliability in a real-time system is determined by the ability of the system to meet such task deadlines as is necessary for the correct functioning of the controlled process.

The problem with conventional approaches to evaluating reliability is that they treat reliability as a static, not a dynamic, quantity. Conventional approaches assume that failure is caused exclusively by the failure of individual hardware or software components. This approach to reliability ignores the dynamic component; namely, that the computer system is only reliable if it does not cause failure for the application (irrespective of the actual number of processors which are still operational).

Central to any reliability evaluation is the definition of an appropriate set of reliability measures. Measures are yardsticks by which reliability is expressed. In other words, reliability measures act as filters, imposing a scale of values that determines which factors are important for "reliability" and which are not.

* This work was supported in part by DARPA, under contract F30602-96-1-0341, order E349.

Currently-used reliability measures for fault-tolerant systems fall in one of two categories.

- *Computer-Centric:* All traditional reliability measures are computer-centric. They focus on the computer in isolation to anything else. The system is defined as being in one of several states, and Markov models are usually used to model the transitions from one state to another. A subset of the states is defined as the failed states, and unreliability is the probability that the system, over a given period of operation, enters one of these states. In traditional models, failed states are entered either after a large number of hardware failures have occurred, leaving the system with insufficient computational capacity, or when there is "coverage failure," i.e., a failure that goes uncaught and uncorrected, and which causes the overall failure of the system. Examples include traditional reliability, availability, and throughput, together with variations such as performance-related reliability measures [1].
- *Application-Centric:* An application-centric measure starts with the premise that performance and reliability can only be meaningfully defined within the context of the application. An application-centric measure would start by considering what the computer needs to do in order to meet the needs of the application. The ability of the computer to meet the application's needs is the application-centric measure of the computer. We will provide some examples in the next section.

Performance measures are used in two related ways. One is to allow a comparison of multiple designs or systems for a given application. The second is to provide an interface by which the designer of the controlled process in which the computer is to be embedded (e.g., aircraft, spacecraft, etc.) can communicate to the computer designer the needs of the application in a form that is intelligible to the latter.

The first use of performance measures is fairly obvious, and is indeed the standard one; the second needs some elaboration. Computer engineers are not trained in control system terminology, and require the needs of the application to be translated to them in terms that they can understand. A good measure of real-time performance or dependability should express the control-theoretic needs of the application in a way that is meaningful to the computer designer. As an example, imagine that there is some cost function by which the impact on the application of a given computer response time (for each task) can be quantified. We shall see an example of this in Section 2. In defining the cost function associated with each control task, the control engineer quantifies the relationship between the response time for each task and the consequent performance of the controlled process, The computer engineer, upon receipt of this information, does not have to be concerned about the control-theoretic foundations of the connection between the controlled process and the embedded computer; the cost functions (and associated hard deadlines) of the various tasks are all that are required.

While application-centric measures have obvious advantages, they have two related disadvantages. The first is that one requires very specific information about the application in order to compute these measures. At an early stage of design, this information may not be available. The second disadvantage is that, by their very nature, application-centric measures are very application-specific. They express the capabilities of the computer entirely with respect to a given application. One cannot directly infer from this the generic ability of a computer to perform in other real-time applications.

The need therefore exists for real-time measures that straddle the middle ground between the computer-centric and the application-centric measures. Such measures should be gracefully degrading with respect to information. That is, as more information is

made available about the application, they should become more application-specific. In the absence of any information about the application, they should express the attributes of the computer that render it capable of running real-time applications. As the amount of information increases, they should become increasingly focused on expressing how the capabilities of the computer meet the particular needs of that application.
In this paper we suggest such a measure, namely, the ability of a computer to handle load surges. We claim that this ability can be used as an indicator of the quality of the following:

- Task assignment and scheduling algorithms.
- Ability of the system to handle a load surge following either an emergency condition or a phase change in the application's mission.
- Procedure followed for the reconfiguration of the hardware upon a failure occurrence.
- Procedure for reassigning and rescheduling tasks upon the failure of one or more processors.

This paper is organized as follows. In Section 2, we provide a brief literature survey of application-centric performance measures. This is followed in Section 3 by a description of the proposed surge-handling measures. Illustrative examples are provided in Section 4, and Section 5 contains some discussion and conclusions regarding the new measures.

2 Prior Work

Not much work has been reported on measures specifically meant for real-time systems. For a survey, see [3, 4]. We describe here two efforts in that direction.

2.1 Performability

This measure was introduced by Meyer [7, 8]. The application is defined as having a set of *accomplishment levels*, which are levels of performance which can be distinguished from one another by the user. We list what the computer must do in order to allow the application to meet each of these accomplishment levels, $A_1, A_2, \cdots, A_n$. The computer can then be modeled to find the probability P_i that it can perform at a level that allows the application to deliver accomplishment level A_i. The vector of probabilities, $P_1, P_2, \cdots, P_n$, is the performability of the computer. See [8] for a detailed example.

2.2 Cost Functions and Probability of Dynamic Failure

This measure was introduced by Krishna and Shin [2, 9, 10, 11], and was designed explicitly for use in embedded systems which handle industrial or other control processes. In such systems, the computer is in the feedback loop of the controlled process, and the computer response time is a component of the feedback loop delay. From elementary control theory, we know that such delays in the feedback loop have a detrimental effect on the quality of the control provided. This can be quantified and used to measure the level of performance provided by the computer. More precisely, we can identify hard deadlines for the tasks by calculating the response time delays that lead to a loss of stability of the controlled process. Even if deadlines are not missed, the cost of having a certain response time can be computed by quantifying the extent to which the performance of the controlled process has degraded. See [4] for further details.

3 The Measures – Definitions

We claim that the survival of a real-time system depends, to a large extent, on its ability to successfully handle unexpected surges in the workload. Such a surge can arise from any of the following causes:

- A change in mission phase, where one set of tasks is replaced by another set. During the period of transition, there can be a surge in the workload, with new tasks having to be run before all of the old tasks have been completed.
- An emergency situation that requires additional tasks to be run. For example, one may have the onset of instability being detected in some vehicle, following which certain additional work may have to be executed in order to restore stability.
- Failure of one or more processors. This affects the system in two ways: first, the workload which was previously run on the failed processor has now to be remapped onto functional processors. This remapping (which includes deciding which tasks to move to which processors, moving the tasks appropriately, and aligning the memories) imposes a transient load on the system. Secondly, the first iteration of such tasks following a move can have significantly reduced laxity, which appears to the surviving processors as a surge in the workload.

We introduce in this paper two new measures which express the ability of the system to respond to such a surge in the workload. For the purposes of this paper, a surge is defined as an additional workload that is suddenly imposed on the system. A surge may consist of one or more tasks, each with its own deadline. Individual tasks cannot be spread out among multiple processors. The size of the tasks determines the granularity of the surge, and the variation of the deadlines determines its homogeneity.

Our surge measures are the following.

- *Minimum Deadline Measure:* Consider a surge of a given magnitude S, consisting of one or more tasks, all with the same deadline. The Minimum Deadline Measure, *MD(S)*, expresses the minimum deadline necessary for the surge so that the system can handle it without missing any deadlines. The smaller the value of the minimum deadline measure for a given surge, the better the system's surge-handling capability.
- *Recovery Time Measure:* This measure, denoted by *RT(S)*, expresses how quickly the effect of a surge of magnitude S on the system fades away. It measures how much time elapses between the arrival of the surge and the point in time when the task schedule is back to what it would have been if no surge had occurred, and no longer has any memory of the surge.

Note that both our measures are curves, functions of the surge size S, rather than single numbers. *MD(S)*, which measures the ability of the system to handle a surge within some deadline given a certain underlying ambient workload, represents the reserve capacity the system has. *RT(S)*, the Recovery Time measure, determines how soon the system recovers and is no longer vulnerable should a second surge follow the first one.

The surge-handling capacity of a system depends primarily on the task assignment and scheduling algorithms as well as on the system architecture. Task assignment and scheduling algorithms determine how effectively the available processing capacity can be harnessed by the workload. The task assignment algorithm can determine how the load is spread out among the processors. For example, some assignment algorithms attempt to balance the load. This can pose difficulties for surges of large granularity, since if the surge consists of a single indivisible task, no one processor may be available which can

execute it on time. Other assignment algorithms attempt to utilize as few processors as possible, thus leaving other processors free to handle even large surges. The latter type of assignment algorithm must be used with caution, however, since it leaves little room for handling variations in execution time demands of the ambient (non-surge) workload. An example comparing such two task assignment algorithms is presented in the next section.

The uniprocessor scheduling algorithm used to determine when to run tasks assigned to individual processors has a similarly large impact on the surge-handling capability. Such algorithms differ in their ability to effectively utilize available processing capacity, and thus differ in their ability to handle surges. In the next section, we will see examples comparing the Earliest Deadline and Rate Monotonic scheduling algorithms.

The system architecture is another important factor in determining surge-handling capacity. To begin with, the architecture governs the raw underlying computational capacity of the system. Also, the interconnection network topology and communication protocol determine the speed – and overhead – with which tasks can be moved from a failed processor to a new one, thus affecting the size of the surge.

The distinction between performability and cost functions on the one hand, and the newly introduced surge-handling measures on the other, should now be fairly obvious. The former are application-centric measures which require detailed – and precise – information about the application before they can be formulated. It would be very difficult to compute these measures either for a generic case, where the focus is not so much on a single application but on a set of possible applications, or when information about the application is incomplete. The new surge-handling measures are not so much application-*centric* as they are application-*sensitive*, in that while their interpretation can be from the point of view of the application (i.e., how much surge handling capability is required by a specific application), they can be computed without detailed information about the dynamics of the controlled process. Further discussion regarding this distinction appears in the final section of this paper.

4 The Measures – Illustrations

To illustrate the use of the two surge-handling measures defined in the previous section for assessing some system attributes, we selected a real-time embedded system comprised of n processors connected through some interconnection network. The ambient workload consists of m periodic tasks, where task k has a period of P_k, an execution time of E_k, a deadline $D_k = P_k$, and an arrival time (of the first iteration of task k) equal to T_k. The load imposed on the system by task k is measured by $U_k = \frac{E_k}{P_k}$ $(k = 1, ..., m)$.

Scheduling of tasks to processors can be done either by using bin-packing, i.e., balancing the load of the n processors, or by the "first-fit" method, which fills each processor up to capacity before moving to the next one. After allocating the m tasks to the n processors, processor i has $m^{(i)}$ tasks, with execution times $E_k^{(i)}$ and periods $P_k^{(i)}$ $(k = 1, ..., m^{(i)}, i = 1, ..., n)$.

At time T_s, the system experiences a surge of size S, with a deadline of D_s. Such a surge may arise either from the arrival of aperiodic tasks or from tasks that have been displaced because their processor has failed, and which must therefore be moved to

other, functional, processors. The surge is divided among the n processors by equalizing the loads of the processors as much as possible. S is divided into $S^{(1)}, S^{(2)}, \cdots, S^{(n)}$ where $S^{(i)}$ is assigned to processor i and $\sum_{i=1}^{n} S^{(i)} = S$.

The order of execution of tasks within a processor is either Rate Monotonic (RM), or follows the Earlier Deadline First (EDF) rule [4, 6] In the RM algorithm, periodic tasks are assigned a static priority, which is proportional to the inverse of their periods. The EDF algorithm, as its name implies, executes the pending task with the earliest deadline. Both algorithms are preemptive.

Surge handling is successful if neither the surge, nor any ambient task, miss their deadlines. We will assume the worst case scenario, i.e., $T_1 = T_2 = \cdots = T_m = T_s = 0$. We will also assume, without loss of generality, that $P_1 \leq P_2 \leq \cdots \leq P_m$.

4.1 The Minimum Deadline Measure

The minimum deadline for a given surge clearly depends on the specific uniprocessor scheduling algorithm employed by each of the processors. We will demonstrate its calculation for the two well-known task scheduling algorithms: the Rate Monotonic (RM) and the Earliest Deadline First (EDF) scheduling protocols.

To calculate the Minimum Deadline measure for a given processor i, following the EDF scheduling protocol, define:

$\alpha_k^{(i)}(\tau) = \left\lfloor \frac{\tau}{P_k^{(i)}} \right\rfloor$ where $\lfloor x \rfloor$ the largest integer smaller than or equal to x.

$\alpha_k^{(i)}(\tau)$ is the number of iterations of task k whose deadline occurs prior to or at time τ. Denote by $MD^{(i,EDF)}(S^{(i)})$ the minimum deadline necessary for a surge of size $S^{(i)}$ arriving at processor i at time $t = 0$, then

$$MD^{(i,EDF)}(S^{(i)}) = Min \left\{ t \;\middle|\; \begin{array}{l} \sum_{k=1}^{m^{(i)}} \alpha_k^{(i)}(t) E_k^{(i)} + S^{(i)} = t \text{ and} \\ \sum_{k=1}^{m^{(i)}} \alpha_k^{(i)}(\tau) E_k^{(i)} + S^{(i)} \leq \tau, \tau = j P_l^{(i)} \textit{ for any integers } j, l \\ \text{where } t < j P_l^{(i)} \leq \prod_{k=1}^{m^{(i)}} P_k^{(i)} \end{array} \right\}$$

The minimum deadline for the whole system of n processors, for a surge of size S and following the EDF scheduling protocol, is

$$MD^{(EDF)}(S) = \underset{1 \leq i \leq n}{Max}\, MD^{(i,EDF)}(S^{(i)})$$

To calculate MD for the RM scheduling protocol, define for a given processor i,

$\beta_k^{(i)}(\tau) = \left\lceil \frac{\tau}{P_k^{(i)}} \right\rceil$ where $\lceil x \rceil$ is the smallest integer greater than or equal to x.

$\beta_k^{(i)}(\tau)$ is the number of iterations of task k which arrived at processor i prior to (but not at) time τ.

In addition, denote: $h^{(i)}(\tau) = max \left\{ k \mid P_k^{(i)} \leq \tau \right\}$ ($h^{(i)}(\tau) = 0$ if $\tau < P_1^{(i)}$).

$h^{(i)}(\tau)$ is the index of the last task in processor i whose period is not greater than τ. then,

$$MD^{(i,RM)}(S^{(i)}) = Min \left\{ t \;\middle|\; \begin{array}{l} \sum_{k=1}^{h^{(i)}(t)} \beta_k^{(i)}(t) E_k^{(i)} + S^{(i)} = t \text{ and} \\ \sum_{k=1}^{h^{(i)}(P_l^{(i)})} \beta_k^{(i)}(\tau) E_k^{(i)} + S^{(i)} \leq \tau, \tau = jP_l^{(i)} \text{ for any integers } j,l \\ \text{where } t < jP_l^{(i)} \leq \prod_{k=1}^{m^{(i)}} P_k^{(i)} \end{array} \right\}$$

and

$$MD^{(RM)}(S) = \max_{1 \leq i \leq n} MD^{(i,RM)}(S^{(i)})$$

We performed some numerical calculations to demonstrate the use of the MD measure for comparing different system attributes. In all our numerical calculations, unless stated otherwise, we used the randomly selected set of values shown in Table 1. The number of processors is $n = 8$ and the number of tasks is $m = 24$. The arrival times are $T_i = T_s = 0$.

Task No.	1	2	3	4	5	6	7	8	9	10	11	12
Period	10	12	12	13	14	15	16	16	17	17	18	18
Execution Time	3	4	2	4	4	1	5	3	1	1	4	4
Load	.30	.33	.17	.31	.29	.07	.31	.19	.06	.06	.22	.22

Task No.	13	14	15	16	17	18	19	20	21	22	23	24
Period	18	19	19	19	20	20	20	20	20	20	21	24
Execution Time	3	5	5	4	6	3	2	5	5	6	7	8
Load	.17	.26	.26	.21	.30	.15	.10	.25	.25	.30	.33	.33

Table 1. The periods, execution times and loads of the 24 tasks.

Figure 1 depicts the Minimum Deadline for a given surge for the RM and the EDF task scheduling algorithms. We clearly see that EDF is superior, allowing the system to successfully handle surges with smaller deadlines.

In Figure 2 we illustrate the effect of increasing the number of processors in the system. The same 24 periodic tasks as in Figure 1 are assumed here and the EDF scheduling algorithm is employed. Clearly. the larger the number of processors the faster the surge handling, but the marginal advantage of increasing the number of processors decreases.

In the previous two figures we have assumed that the surge appears in the worst possible time instant when all 24 tasks are waiting to be executed, i.e., at t=0. In Figure 3 we examine the dependence of our measure on the exact time instant when the surge

occurs. The same 24 periodic tasks are assumed here and the EDF scheduling algorithm is employed.
Figure 4 depicts the combined effect of varying both the scheduling algorithm and the number of processors on the surge handling capability of the system.

4.2 The Recovery Time Measure

RT(S), the recovery time for a given surge S, does not depend on the scheduling algorithm within the processor, as long as the procedure is "work conserving", i.e., the processor is never idle when there are tasks to be executed. *RT(S)* can be calculated as follows.

As before, define for a given processor i: $\beta_k^{(i)}(\tau) = \left\lceil \frac{\tau}{P_k^{(i)}} \right\rceil$.

The recovery time from a surge of size $S^{(i)}$ for processor i is

$$RT^{(i)}(S^{(i)}) = min \left\{ t \;\middle|\; \sum_{k=1}^{m^{(i)}} \beta_k^{(i)}(t) E_k^{(i)} + S^{(i)} = t \right\}$$

and the recovery time for the n processor system is

$$RT(S) = \max_{1 \le i \le n} RT^{(i)}(S^{(i)})$$

Figure 5 depicts the Recovery Time as a function of the surge for an eight processor system with the same 24 periodic tasks as before. The scheduling algorithm employed is EDF and the surge is assumed to occur at t=0. We compare two task allocation algorithms, namely the first-fit and the bin-packing algorithms. We also consider two values of surge granularity, where the surge is divided into either three or five indivisible tasks. We can see that for the higher surge granularity (surge is divided into three tasks) bin-packing is preferred, while for the lower granularity first-fit is better.
Another important use for the Recovery Time measure is for comparing different fault-recovery procedures. The overhead involved in recovering from a fault can be considered a surge, and the better the recovery procedure, the shorter the surge recovery time. In this case, the *RT* measure can be used for comparing the different attributes of the recovery procedure. In Figure 6, the effects of the checkpointing interval and the checkpointing overhead are investigated. An intermittent fault is assumed to occur at processor 2 at time 0. Figure 6 depicts the Recovery Time as a function of the checkpointing interval for two values of the checkpointing overhead, namely 1 and 2 time units. Clearly, there is an optimal value of the checkpointing interval, and it is larger for the larger value of the overhead.

5 Discussion and Conclusions

Traditional performance measures have tended to be either totally computer-centric or totally application-centric. The problem with the former is that they do not take the needs of the application into account. While the latter type of measure is ideal when perfect information is available about the application, it is useless when the application

is still under development, and full information about it is not available. By contrast, the surge-handling measures introduced here can be computed for a system using the current state of knowledge about the ambient workload. Another case when application-centric measures are useless is when we want to characterize the computer, not in the context of a specific application, but with respect to its general suitability for real-time applications. On the other hand, it is possible to characterize the surge-handling capability of the system for any ambient workload and any architecture. Recently, efforts have been made by several research teams to build standard real-time benchmarks, most notably by Mitre and Honeywell corporations. These benchmarks can be used to define the ambient workloads in terms of which the surge-handling measures can be evaluated. The surge-handling measures can also be used to evaluate the quality of real-time operating systems, especially their task assignment and scheduling algorithms. Other features that can be evaluated using these measures are the interconnection topology and the communication protocols (since they determine the costs associated with moving tasks) as well as the failure recovery procedures, including the checkpoint placement strategy [5].

References

1. M. D. Beaudry, "Performance-Related Reliability Measures for Computing Systems," *IEEE Trans. Computers*, Vol. C-29, 1978.
2. C. M. Krishna and K. G. Shin, 'Performance Measures for Multiprocessor Controllers," in A.K. Agrawala and S.K. Tripathi, *eds., Performance '83*, 1983.
3. C. M. Krishna and K. G. Shin, "Performance Measures for Control Computers," *IEEE Trans Automatic Control*, Vol. AC-32, 1987.
4. C. M. Krishna and K. G. Shin, *Real-Time Systems*, New York: McGraw-Hill, 1997.
5. C. M. Krishna, K. G. Shin, and Y.-H. Lee, "Optimization Criteria for Checkpointing," *Communications of the ACM*, Vol. 27, No. 10, 1984.
6. C. L. Liu and J. W. Layland, "Scheduling Algorithms for Multiprogramming in a Hard Real-time Environment," *Journal of the ACM*, Vol. 20, 1973, pp. 46–61.
7. J. F. Meyer, "On Evaluating the Performability of Degradable Computing Systems," *IEEE Trans. Computers*, Vol. C-29, 1980.
8. J. F. Meyer, D. G. Furchtgott, and L. T. Wu, "Performability Evaluation of the SIFT Computer," *IEEE Trans. Computers*, Vol. C-29, 1980.
9. K. G. Shin and C. M. Krishna, "Characterization of Real-Time Computers," *NASA Contractor Report 3807*, August 1984.
10. K. G. Shin and C. M. Krishna, "New Performance Measures for Design and Analysis of Real–Time Multiprocessors," *Journal of Computer Science and Engineering Systems*, Vol. 1, pp. 179–192, October 1986.
11. K. G. Shin, C. M. Krishna, and Y.-H. Lee, 'A Unified Method for Characterizing Real–Time Computer Controller and its Application,"*IEEE Transactions on Automatic Control*, Vol. AC–30, No.4, April 1985, pp. 357–366.
12. K. Yu and I. Koren, "Reliability Enhancement of Real-Time Multiprocessor Systems through Dynamic Reconfiguration," *Fault-Tolerant Parallel and Distributed Systems*, D. Pradhan and D. Avresky (Editors), pp. 161-168, IEEE Computer Society Press, Los Alamitos, CA, 1995.

This article was processed using the LaTeX macro package with LLNCS style

Appendix 5

Synthesis of Interconnection Networks: A Novel Approach

Vijay Lakamraju, Israel Koren and C.M. Krishna

Department of Electrical and Computer Engineering
University of Massachusetts, Amherst 01003
E-mail: {vlakamra,koren,krishna}@ecs.umass.edu

Abstract

The interconnection network is a crucial element in parallel and distributed systems. Synthesizing networks that satisfy a set of desired properties, such as high reliability, low diameter and good scalability is a difficult problem to which there has been no completely satisfactory solution.

In this paper, we present a new approach to network synthesis. We start by generating a large number of random regular networks. These networks are then passed through filters, which filter out networks that do not satisfy specified network design requirements. By applying multiple filters in tandem, it is possible to synthesize networks which satisfy a multitude of properties. The filtered output thus constitutes a short-list of "good" networks that the designer can choose from. The use of random regular networks was motivated by their surprisingly good performance with regard to almost all properties that characterize a good interconnection network.

Experimental results have shown that this approach is practical and powerful. In this paper we focus on the generation of networks which have low diameter, good scalability and high fault tolerance. These generated networks are shown to compare favorably with several well-known networks.

[1]This research was supported in part by DARPA and the Air Force Research Laboratory under Grant F30602-96-1-0341. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Projects Agency, the Air Force Research Laboratory, or the US Government.

1. Introduction

Interconnection networks (ICNs) are as much a determinant of performance and dependability in a parallel or distributed system as the processors themselves. The network impacts the cost of the architecture and the cost of communicating between processors, as well as system reliability and the extent to which the system can degrade gracefully under processor or link failures.

This paper describes a new approach to the synthesis of interconnection networks for parallel and distributed systems. The distinguishing features of our technique are that it can be tailored to the specific performance and fault-tolerance measures of interest to the designer, and that it can be used even by those who are not experts in interconnection networks. It is especially useful when seeking to synthesize a network that performs well with respect to *multiple* performance measures. For example, a designer may place a high premium on both scalability and network resilience, while simultaneously needing to constrain the degree of the network. It can also be used to study tradeoffs among several performance or dependability parameters.

A vast literature on interconnection networks exists. Networks such as the hypercube, shuffle-exchange, Banyan, bus, chordal ring, tree and others, have been extensively studied [8, 9]. However, much less has been reported on the problem of synthesizing a network to meet specific performance and reliability criteria.

In our approach, the designer specifies the performance measures of interest. These may be commonplace measures such as bandwidth, diameter, connectivity, or more exotic measures like diameter stability in the face of failure, the extent to which the network splinters as node and link failures accumulate, or scalability. A large number of random regular networks of the desired size are then generated and passed through a bank of *filters*. Each filter is associated with a per-

formance requirement. The filters identify a subset of networks which have the desired performance with respect to the specified measures. This subset constitutes a short-list of networks from which the designer can choose.

The usefulness of this approach rests on its efficiency. That is, the number of random networks one has to generate before obtaining a useful short-list of "good" networks. This problem does not readily yield to theoretical analysis, and must be studied by simulation experiments. We have found, through extensive experimental work, that our technique is surprisingly efficient.

The rest of the paper is organized as follows. In the next section, we briefly review various desirable properties of ICNs. In Section 3, we describe our random network generation algorithm and the filtering process. Section 4 provides extensive experimental evidence to the good performance of random regular networks. It also shows the effectiveness of the filtering approach through examples. Section 5 summarizes our findings and discusses future work.

2. Preliminaries

A good interconnection network is characterized by a number of desirable properties. Some of these are listed below:

- *Small internodal distances.* One factor in the communication delay is the node-pair distances. The greater the average node-pair distance, the greater the time a message will spend in the network, the greater the energy consumed in delivering it, and the greater the chances of network congestion.

- *Small, fixed degree.* Each physical connection costs money and a small degree corresponds to reduced wiring and fewer I/O interfaces. Furthermore, if the degree is constant over all nodes, then only one basic node design may be necessary.

- *Good fault tolerance.* Many parallel or distributed systems are used in applications requiring levels of reliability that can only be achieved by making the system fault-tolerant. There are many measures of fault-tolerance: we list below a partial list of the more useful network measures.

  - Probability of network disconnection.
  - Diameter stability, i.e., how the network diameter is expected to increase as nodes or links fail.
  - Stability of the average node-pair distance as nodes or links fail.
  - How the network splinters after it gets disconnected: is it more likely to splinter into one large component which is still useful, and several small and useless components, or will all the components be too small to be useful?

- *Easy construction and good scalability.* It should be possible to construct a network of any desired size. Further, adding a few nodes to the network should not cause drastic changes in such properties as diameter or average node-pair distance. A scalable ICN should be able to accommodate small increases in size rather than only large increases.

- *Embeddability.* Some algorithms are designed to run well on certain topologies, i.e., those that map well to the communication pattern of the application. A good network should be able to embed a wide range of topologies with low dilation, thus ensuring that a large number of algorithms will run efficiently on the selected ICN.

- *Easy routing algorithms.* It is advantageous to have a simple routing algorithm, for example, one that requires only the knowledge of the destination address. Routing algorithms can have a big impact on congestion and power requirements. Networks that facilitate the use of such simple algorithms are preferable.

These measures will vary in importance from one application to another. For example, space applications may require massive levels of fault-tolerance and low power consumption, while not placing a large premium on scalability.

Interconnection networks can be represented as graphs in which the vertices correspond to processors and the edges to communication links. In this paper we use the terms networks and graphs interchangeably. We consider only undirected graphs and the size of a network refers to the number of vertices in the graph. In this paper, we mainly concentrate on networks of degree 3 and 4, though our approach is not restricted to these degrees. For comparing the performance of different measures among degree-3 networks, we use the following topologies: shuffle exchange networks [16], cube connected cycles(CCC) [14], chordal rings of degree 3 [1], Moebius trivalent graphs [12] and multi tree structures(MTS) of degree 3 [2]. In the degree-4 catogory, we use meshes, torii, chordal rings of degree 4 [7] and the wrapped butterfly networks. In the next section, we describe our approach to synthesizing networks which meet the designer's requirements.

3. Approach

Our approach to network synthesis consists of a two-step process: first, the generation of a large set of random regular networks and second, the isolation of just the right ones through a process of filtering.

3.1. Generating Random Regular Networks

We use the following definition of random regular graphs:

Definition 3.1 *A random regular connected graph of size n and degree d is a d-regular connected graph in which node pairs connected by an edge are selected at random.*

Random regular graphs of n nodes and degree d are generated as follows. We start with a set of n isolated nodes. Edges are placed between node pairs selected at random. This process continues until all the nodes in the network satisfy the following two requirements: *(i)* the degree of all the nodes is the same and equal to the specified value, d and *(ii)* no pair of nodes is joined by more than one edge, and no self-loops exist. Finally, the generated network is tested for connectedness. Algorithm 1 contains the pseudocode used to generate random regular networks.

Algorithm 1
generate_regular_random_network(size,degree,seed)

```
1:  A ⇐ {1, ..., n}
2:  repeat
3:    Randomly pick two nodes, u and v, from set A
4:    if ((u ≠ v) and edge(u,v) not already present)
      then
5:      Add edge(u,v) to the adjacency matrix
6:      update A by removing nodes whose degree has
        been satisfied
7:    else if (size(A) = 1) or
          (nodes in A form a fully connected subgraph)
      then
8:      discard and start all over again
9:    end if
10: until size(A) = 0
11: check for connectedness
12: if graph not connected then
13:   discard and start all over again
14: else
15:   return adjacency matrix
16: end if
```

A is the set of all nodes whose degree has not been satisfied and is initialized on line 1 to the set of all n labelled nodes. Lines 4–6 ensure that the two conditions stated above are met and lines 7–8 ensure that the algorithm does not loop infinitely. If, during construction, the nodes in A form a fully connected subgraph, then no matter which two nodes are picked, the connecting edge will always be superfluous. No attempt is made to backtrack from this situation, and so the current adjacency matrix is discarded and a new one generated.

The above algorithm generates a random regular network each time it is called with a different seed value. Note that under some conditions, such as those on lines 7 and 12, the network that is being generated needs to be discarded and the generation restarted. To estimate the runtime required to generate a valid graph, we generated a large number of random graphs for various network sizes and calculated the average runtime for each network size and for different degree networks. Results shown in Figure 1 were obtained on a 500MHz Pentium having 256MB of memory. It was observed that networks of even 2048 nodes could be generated in less than a second using this algorithm. This shows that the generation algorithm can output a random regular graph in reasonable time. It was also observed that the check for graph connectedness (line 12 of the algorithm) was almost always satisfied. It is also important that the generated networks are non-isomorphic to each other; otherwise the filtering process will not make sense. To find out how many of the generated networks are non-isomorphic, we checked the isomorphism between all pairs of networks and observed that more than 99% of the networks were non-isomorphic to each other. All these results show that the generation algorithm provides a cheap and versatile method for producing the "raw" material for the filtration process. To get a better idea of the number of distinctive networks that can be generated with size n and degree d, see [15].

3.2. The Filtering Process

The raw material for the filtering process is the set of random graphs generated. Filtering consists of identifying those networks which have the properties desired by the designer.

We use one filter for each requirement to be satisfied. Typically each requirement is associated with a single performance measure or a set of measures. A filter consists of two parts: the evaluation part calculates the value of the measure associated with the requirement, and the checking part compares the value of the measure with a *threshold* specified by the requirement. For example, if the requirement was a diameter no greater

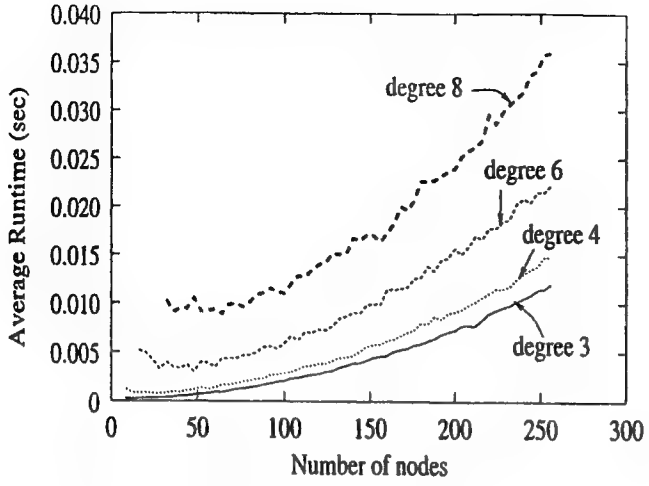


Figure 1. Average runtime of the generation algorithm

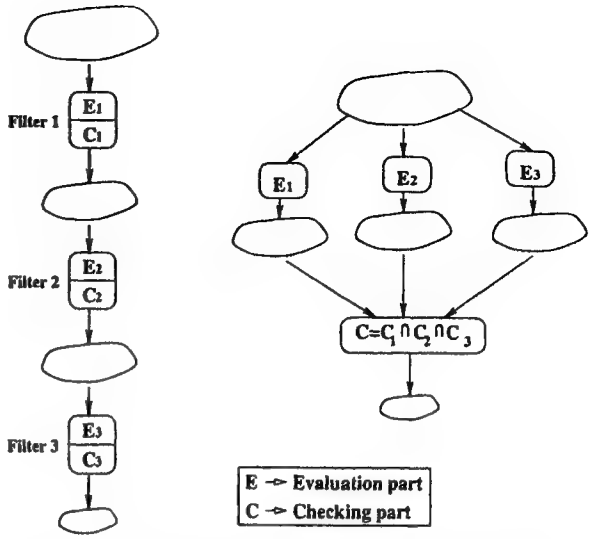


Figure 2. Sequential and Parallel Filtering

than k, then the evaluation part computes the diameter of the network and the checking part checks whether this requirement has been met. Each filter takes as input a set of random networks and outputs only those that pass the checking part. The output of one filter is used as input to the next. The filters are arranged sequentially one after the other in decreasing priority order of the measures they represent. The output at the end of the entire filtering process depends on the threshold values that have been set for each filter. If the filtering process produces no output, the designer will have to refine the threshold values. The threshold of a higher-priority filter should not be relaxed before that of every lower-priority filter has been relaxed to the maximum allowed extent. The key feature of this filtering approach is its versatility, as the set of selected filters and their order is determined by the specific application requirements.

The evaluation part is typically much more time-consuming than the checking part. In order to speed up the filtering process, the evaluation corresponding to each of the filters can be carried out in parallel and a single checking part that combines the checking parts in all filters used to sift out networks that comply with all the requirements. This approach is called parallel filtering, compared to the sequential filtering that we described earlier (Figure 2). Note that the time taken in the case of parallel filtering is bounded by the maximum evaluation time among the filters, and evaluation is carried out for all the input networks. In sequential filtering, the threshold determines the number of networks that pass through at each stage. If a stringent threshold is used, a smaller number of networks pass through and this greatly impacts the time spent in the remaining filters. Thus, the time taken in the case of sequential filtering is dependent on the threshold set in each filter.

Some implementation details are worth mentioning. One need not store the adjacency matrices of all the input networks because they can be regenerated easily and quickly using the seed value. So, only the seed values used to generate the random networks need to be stored. Also, thresholds can be specified in relative terms rather than using an absolute threshold value (for example, take the best 5% of the input networks), although this requires sorting the input networks according to the value obtained from the evaluation part.

4. Experimental Results

In this section, we demonstrate the efficiency of our filtering approach by considering the synthesis of ICNs with required diameter, scalability and fault-tolerance characteristics.

4.1. The Diameter Filter

The diameter, Δ, which is the maximum of the node-pair distances, provides an upper-bound on the inter-task communication time, in terms of hops, and can be a decisive factor in application runtime. The problem of constructing a network of a given size and degree with the smallest possible diameter has been the focus of much research [4, 13]. While the diameter of random graphs has also been studied, the published results tend to be of an asymptotic nature, valid as

the size of the graph approaches ∞. These asymptotic results provide little guidance for graph sizes that are of practical interest. In order to evaluate the diameter of random regular networks, we generated random networks sized between 8 and 256 nodes and with degrees ranging from 3 to 6 and calculated their diameters. For each size and degree, 1000 random networks were generated and the ones with the least diameter were selected. Figure 3 shows how the diameter slowly increases with size and how it reduces as the degree is increased. These results provide a lower bound to the threshold that can be set for the diameter filter. Figure 4 shows the comparison of the diameter of random networks of degree 3 with other networks of the same degree. The diameters of the networks plotted are the ones with the least diameter as specified in their respective references.

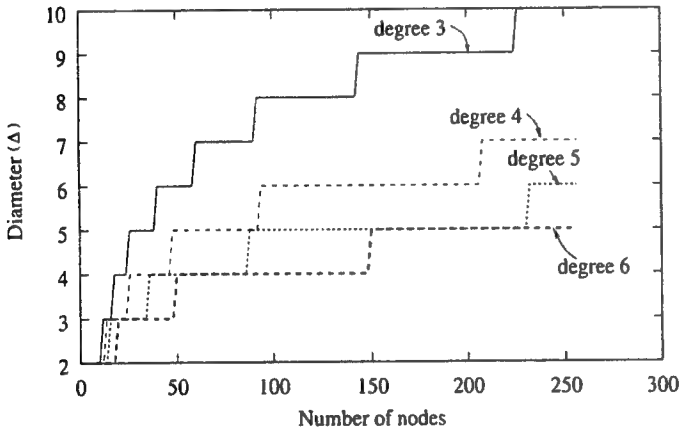


Figure 3. Diameter of random regular networks

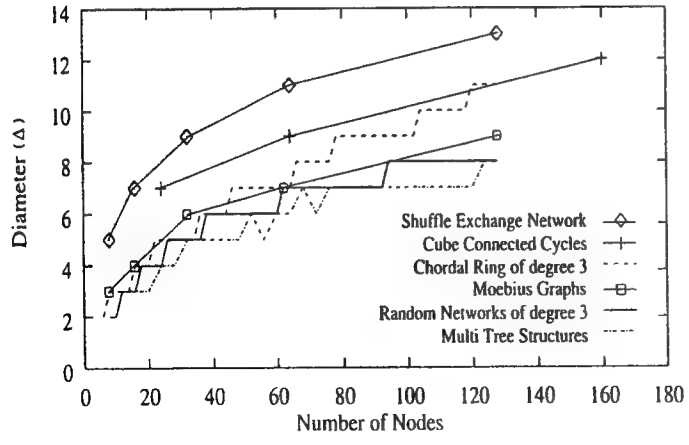


Figure 4. Diameters of different networks of degree 3

From Figure 4 it is clear that random networks perform better than such common ICNs as the mesh and the hypercube, but have greater diameter than some well-crafted ICNs such as the MTS network for some network sizes. However, graphs such as MTS are not as flexible. The MTS network is defined only for certain sizes given by $m * (d-1)^{t-1}$ where m and t are integer parameters[1]. Among networks of degree 4 that we have considered, random regular networks performed the best. It is worthwhile to find out the number of graphs that pass through when the threshold of the diameter filter is set to different values. Figure 5 shows the frequencies of networks of degree 3 that pass through diameter filters whose thresholds have been set at the minimum diameter (as shown in Figure 3).

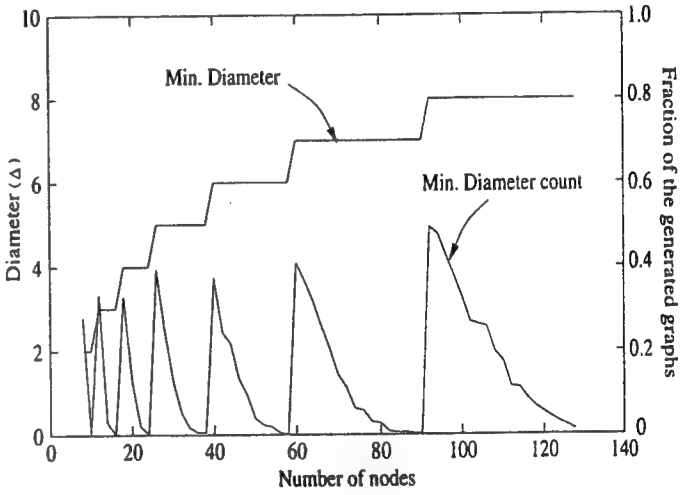


Figure 5. Frequency of minimum diameter random networks of degree 3

Figure 4 is very interesting because it shows that if we generate a sizeable number of random networks and then select the one with the smallest diameter, we will (with a high probability) get a network that is diameter-competitive with most of the interconnection networks described in the literature. It should be pointed out, however, that the size of the random graphs of a given degree and diameter tend to be greater than theoretical bounds, such as the Moore bound[3] or the bound obtained from theoretical studies of random graphs[5, 6].

Further comparisons can be carried out with the entries in the (d, Δ) table[2]. Table 1 shows some of the

[1] Diameters of incomplete MTS networks have not been analyzed as yet.

[2] The (d, Δ) table gives the *state of the art* with respect to a largest known graphs with degree d and diameter Δ [10].

results. The diameter of the random graphs was at most larger by 1 than the corresponding best known diameters. It is worth pointing out that these known networks are constructed by different methods for different degrees and diameters whereas the random networks follow the same simple construction algorithm.

Size of Network	Degree	Diameter	
		Best Known	Random
10	3	2	2
15	4	2	3
20	3	3	4
70	3	5	6
364	4	5	6
532	5	5	6
740	4	6	7

Table 1. Comparison of diameter between best known networks and the best of the random networks generated in our experiments

4.2. The Scalability Filter

Some applications and situations require networks to be scalable. A network is said to have good scalability if the size of the network can be increased with minimum disruption and this does not cause a drastic change in its properties. For reasons of cost, it is better to have the option of small increments since this allows the network to be upgraded to the required size within a particular budget. The hypercube, for example, has poor scalability, in that its size cannot be increased by small increments while still maintaining its structural properties. Random graphs, on the other hand, have good scalability. They can be constructed for all sizes and degrees (as long as $n*d$ is even) and Figure 3 shows that the diameter remains constant for a considerable range of network sizes.

If regularity of the graph must be maintained even after scaling, some edges must be removed and some added to accomodate the new node. The minimum number of edges that must be removed to scale an even degree network by one node is $d/2$ whereas that required to scale an odd degree network by two nodes is $d-1$. Typically, one does not possess the flexibility of adding new nodes anywhere in the network. It may be required to attach the new node adjacent to a given set of nodes. This is typically the case in a fault-tolerant design when a spare processor must serve as a backup for a given set of processors. We define a measure for scalability in this context by the average increase in diameter caused by connecting a new node to all possible designated sets of d nodes. The network is said to have good scalability if its diameter does not increase considerably on average.

Not all randomly generated graphs scale well. To evaluate the performance of random graphs with regard to scalability, we generated 100 random graphs of size 64 and degree 4 and diameter 5. Note that 5 is the minimum diameter obtained for graphs of size 64 and degree 4 as shown in Figure 3. For each network, we then evaluated scalability by selecting sets of four nodes at random and adding a new node adjacent to the designated nodes. If the designated nodes are connected by an edge, then this edge is removed, otherwise edges incident on the designated nodes are selected at random and removed to create connections to the new nodes as shown in Figure 6.

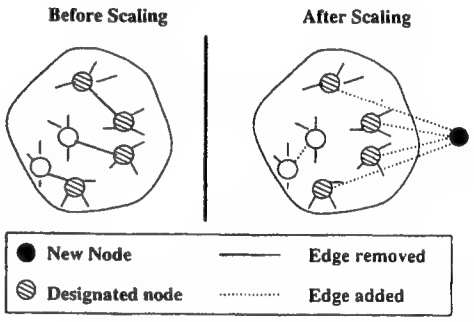


Figure 6. Example of edges that can be removed to accomodate a new node

The diameter is calculated for each set of four nodes and the increase in diameter is averaged over a large number of such runs. Figure 7 shows the cumulative distribution of the increase in diameter for the input graphs. It gives an idea of the threshold that can be set for a scalability filter, e.g., if the best 10% of the graphs are selected then we can expect that the average increase in diameter will be no more than 0.05.

4.3. The Fault-Tolerance Filter

Reliability is an important criterion in the selection of an interconnection network. Measures are required to adequately capture network qualities such as graceful degradation and robustness. Traditional measures, such as connectivity, are worst-case measures and have limited expressiveness. In this paper, we look at the following more expressive measures: the diameter stability, $\Delta(p_f)$, the average node-pair distance stability, $\overline{D}(p_f)$, the probability of disconnection, $\pi_d(p_f)$, and the size of the maximum connected component,

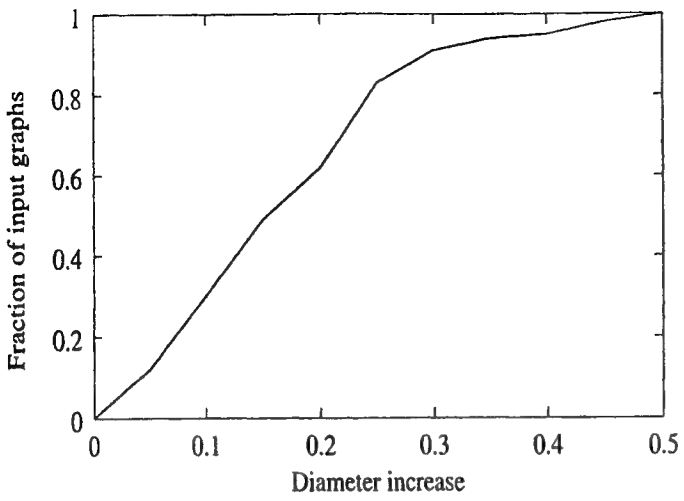
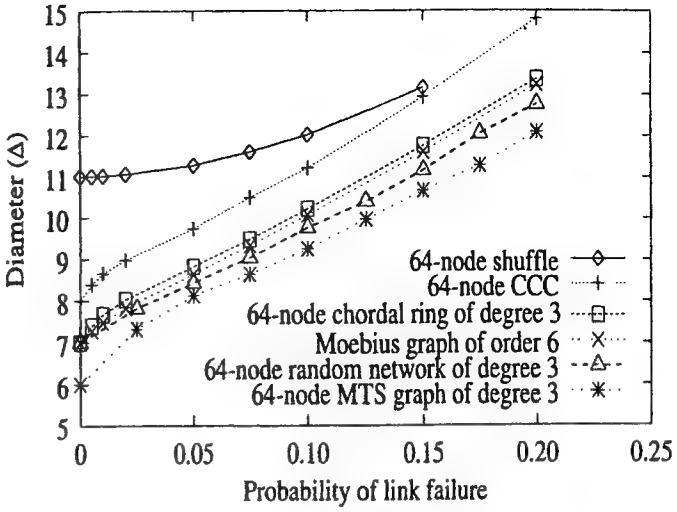
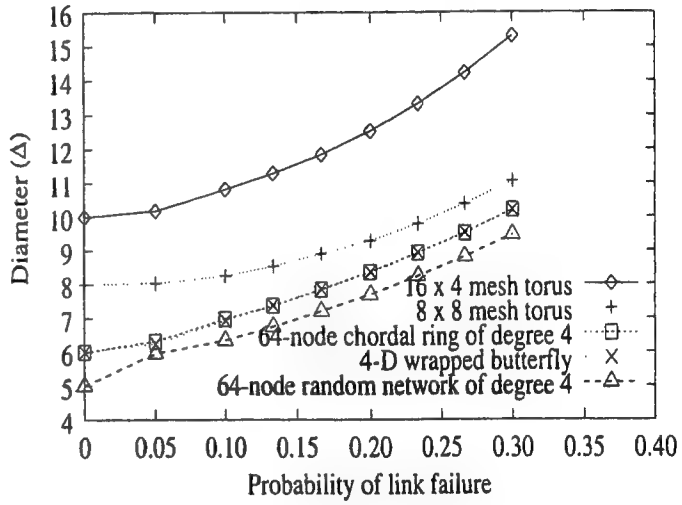


Figure 7. Cumulative frequency of the increase in diameter for random networks of size 64 and degree 3



(a) Comparison among degree 3 networks



(b) Comparison among degree 4 networks

Figure 8. Diameter vs. probability of link failure

$\chi_{max}(p_f)$, all in the presence of link failures occurring independently with probability p_f. These measures were introduced in [11] and some research has already been done in characterizing various networks with respect to these measures. We performed experiments to evaluate the vulnerability of regular random networks with respect to these four measures. We used random networks of size 64 and degree 3 and 4 and compared their performance with other networks of similar size and degree. The network used was chosen at random from the set of minimum diameter networks obtained at the output of the diameter filter.

Figure 8 shows the comparison of diameter stability among degree 3 and degree 4 networks. The random regular networks of degree 4 outperform all the networks in its category whereas in the degree 3 category, it is second-best. Though Figures 9 and 11 show average node-pair distance stability and probability of disconnection for degree 3 networks only, the performance of random networks in the degree 4 category was observed to be the same as in the case of diameter stability. All these results show that random networks perform better than most of their counterparts with respect to fault tolerance as well. Careful examination of the results reveals that networks that are not regular are more vulnerable compared to those that are regular, as can be seen in the case of shuffle exchange networks and meshes.

The fault-tolerance filter that we use is a combination of four filters: one for each of the four measures. Thresholds are typically specified as a scalar value and since each of the fault tolerant measures is given by a vector of values (corresponding to different link(node) failure probabilities), some transformation must be used to convert the vector of values to a single value. If the designer knows the exact value of the link failure probability, then the value of the measure corresponding to that failure probability can be used to do the comparison. However, it may be difficult for the designer to decide on the exact value of the failure probability. One transformation that can be applied would be to use the area under the curve obtained when plotting the values. This transformation assumes that each failure probability in the range

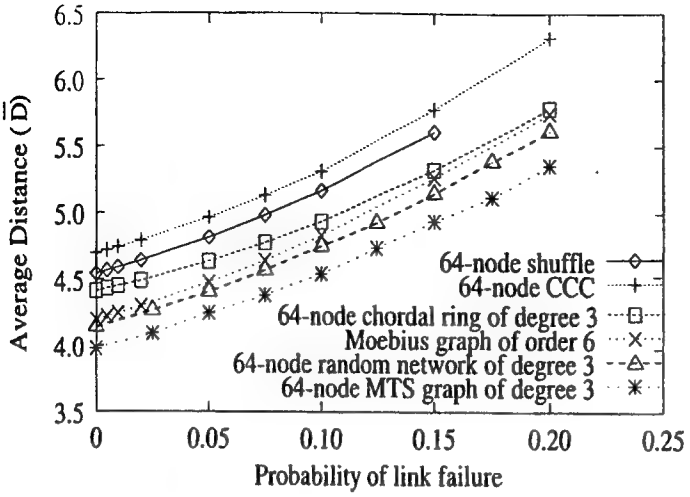


Figure 9. Average node-pair distance vs. probability of link failure

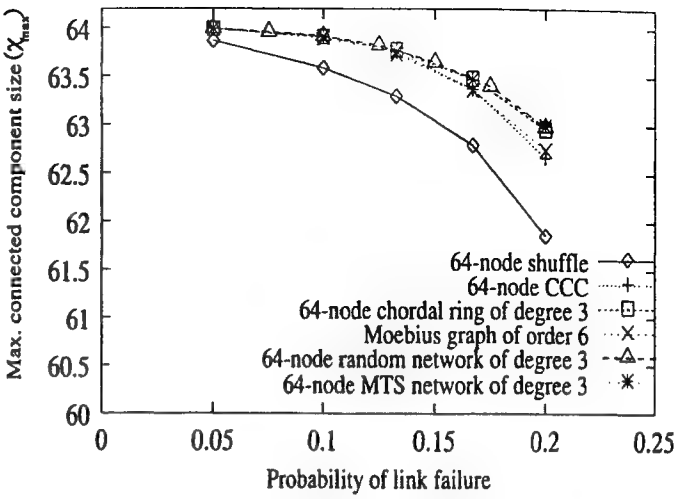


Figure 11. Maximum component size vs. probability of link failure

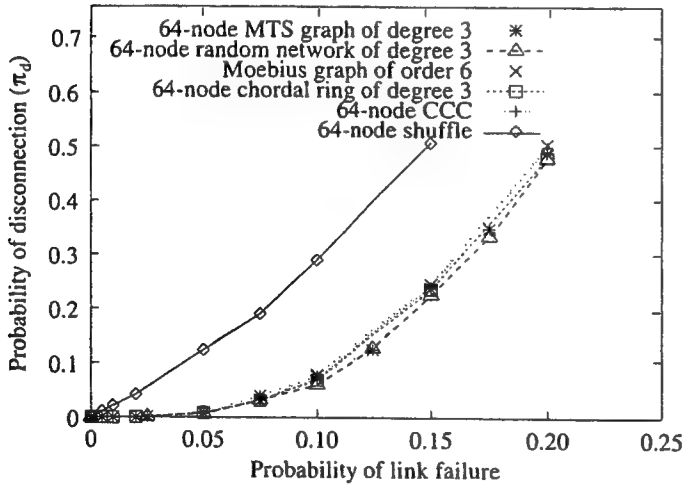


Figure 10. Probability of disconnection vs. probability of link failure

of interest is equally likely. Since this may not be the case in most situations, a more appropriate transformation would be to associate weights with each of the failure probabilities and use a weighted sum of the values corresponding to different failure probabilities.

The output at the end of the set of filters depends on the threshold values that have been set for each filter. A stringent threshold value passes a smaller number of networks through it. If the filter produces no output, the designer would have to refine the threshold values or increase the number of input graphs generated.

Experiments were performed to evaluate the efficiency of the approach by passing random graphs through a multitude of filters. Graphs of size 64 and degree 4 were generated and first passed through a diameter filter and then through a fault tolerance filter. The threshold of the diameter filter was set at 7. The networks obtained at the output of the diameter filter were tested for their fault tolerant characteristics by using two filters, the $\Delta(p_f)$ filter and the $\overline{D}(p_f)$ filter. The range of link failure probabilities of interest to us in this example was $[0.0, 0.2]$. Our initial set consisted of 1000 randomly-generated networks. After passing through the diameter filter, we were left with 33%(=330) of the networks. These networks were then evaluated for the two fault tolerance measures and then ranked according to their performance. The thresholds of the filters were set such that only those input networks that lie among the best 5% pass through it. The number of graphs obtained at the end of the filtering process was a respectable 1.5%(=15). It is important to note this "short-list" contains graphs that are better than most graphs published in the literature with regard to diameter and the two fault tolerance measures.

5. Conclusions and Future Work

Synthesizing networks that satisfy a certain set of performance or fault-tolerance requirements is difficult. In our approach, we generate a large number of random regular networks and filter out those that do not comply with the requirements. The choice of random regular networks was motivated by their ease and flexibility of construction and their surprisingly good prop-

erties. The filtering process consists of filters arranged in tandem, one for each requirement to be satisfied. Each filter removes networks that do not comply with the requirement associated with it. The strength of this approach lies in the versatility and extendability of the filtering step, in that a different set of filters can be used for a different set of requirements and new filters can be added as and when newer measures are developed. We demonstrated the effectiveness of this approach by synthesizing fault-tolerant networks with a small diameter.

Extensions to the current work are ongoing in several directions. Other filters are currently being studied, among them are the filter for embeddability and routability. Other network-generation algorithms are being developed and assessed. A graphical tool to facilitate synthesis of interconnection networks through our approach is also on the anvil.

Acknowledgement

The authors wish to thank Zahava Koren for stimulating discussions and suggestions.

References

[1] B. Arden and H. Lee. Analysis of chordal ring networks. *IEEE Trans. Computers*, C-30:291–295, Apr 1981.

[2] B. Arden and H. Lee. A regular network for multicomputer systems. *IEEE Trans. Computers*, C-31:60–69, Jan 1982.

[3] E. Bannai and T. Ito. On finite moore graphs. *J. Fac. Sci., Tokyo Univ.*, pages 191–208, 1973.

[4] J. Bermond and B.Bollobas. The diameter of graphs - a survey. *Proc. Congressus Numerantium*, 32:3–27, 1981.

[5] B. Bollobas. Random graphs. *Combinatorics Lect. Note Series London Mathematic Soc.*, pages 80–102, 1980.

[6] B. Bollobas and W. L. Vega. The diameter of random regular graphs. *Combinatorica*, 2:125–134, Feb 1982.

[7] K. Doty. New designs for dense processor interconnection networks. *IEEE Trans. Computers*, C-33:447–450, May 1984.

[8] T. Feng. *Editorial Introduction, Tutorial Interconnection networks for Parallel and Distributed Systems.* IEEE Press, Piscataway, NJ, 1984.

[9] R. Finkel and M. Solomon. Processor interconnection strategies. *IEEE Trans. Computers*, 29:360–370, May 1980.

[10] http://maite71.upc.es/grup_de_grafs/table_g.html. d-k table.

[11] V. Lakamraju, Z. Koren, I. Koren and C. M. Krishna. Measuring the vulnerability of interconnection networks in embedded systems. *Proc. First Merged Symposium IPPS/SPDP, EHPC Workshop*, pages 919–924, April 1998.

[12] W. Leland and M. Solomon. Dense trivalent graphs for processor interconnection. *IEEE Trans. Computers*, vol C-31:219–222, March 1982.

[13] J. Opatrny, D. Sotteau, N. Sitaraman and K. Thulasiraman. DCC linear congruential graphs: A new class of interconnection networks. *IEEE Trans. Computers*, C-45:156–164, Feb 1996.

[14] F. Preparata and J. Vuillemin. The cube connected cycles: A versatile network for parallel computation. *Communications of the ACM*, pages 300–309, May 1981.

[15] R. Read. The enumeration of locally restricted graphs. *J. London Math Soc.*, pages 417–436, 1959.

[16] H. Stone. Parallel processing with the perfect shuffle. *IEEE Trans. Computers*, C-20:153–161, Feb 1971.

Appendix 6

APPLICATION-LEVEL FAULT TOLERANCE

A Thesis Presented

by

JOSHUA WILLS HAINES

Submitted to the Graduate School of the
University of Massachusetts Amherst in partial fulfillment
of the requirements for the degree of

MASTER OF SCIENCE IN ELECTRICAL AND COMPUTER ENGINEERING

May 1999

Department of Electrical and Computer Engineering

ABSTRACT

APPLICATION-LEVEL FAULT TOLERANCE

MAY 1999

JOSHUA WILLS HAINES

B.S.E.E., UNION COLLEGE, SCHENECTADY NY

M.S.E.C.E., UNIVERSITY OF MASSACHUSETTS AMHERST

Directed by: Professor Israel Koren

As multiprocessor systems become more complex, their reliability will need to increase as well. In this thesis we propose a novel technique which is applicable to a wide variety of distributed real-time systems, especially those exhibiting data parallelism. We assert that for high reliability, a combination of system-level fault tolerance and application-level fault tolerance works best. In many systems, application-level fault tolerance can be used to bridge the gap when system-level fault tolerance alone does not provide the required reliability. We exemplify this with the RTHT target tracking benchmark and with the ABF beamforming benchmark, and discuss integration with two additional parallel applications.

TABLE OF CONTENTS

LIST OF TABLES

Table

LIST OF FIGURES

Figure

CHAPTER 1

INTRODUCTION

Every computer system in operation today is certain to experience faults during its operable lifespan. Faults may be thought to *arrive* at a system from many sources, including environmental factors, interaction with other systems, interactions with human users, or hardware and software design flaws accidentally built into the system. A system that is able to function reasonably well despite the manifestation of faults such as these is thought to be highly reliable.

1.1 System Reliability

Reliability is one of the most desirable traits of any computer system today, be it a desktop workstation, used for multimedia applications, or a mission-critical server in some capacity. The reliability of a system is linked directly to its ability to either avoid faults or its ability to operate correctly despite the presence of faults [2]. Fault avoidance includes any technique designed to decrease the likelihood of a fault occurring anywhere in the system, while fault tolerance includes any technique to minimize the effects of a fault upon system operation. Most modern computer systems make use of both techniques to some degree.

1.2 Fault Tolerance

In order to better understand the following taxonomy, it should be understood that an *error* is the manifestation of a *fault*. The process of *tolerating* a fault might include detecting the presence of the fault, or an error generated out of the fault, and reacting to that fault or error to hopefully combat its effects. Faults may be detected in many ways. In general, a system could look for either the fault itself, or attempt to detect the error(s) generated from the fault. Timeout schemes could be employed: if a particular component does not respond to a query within certain time, it could be assumed faulty. Data could be checked for accuracy or correctness: input or output data could be compared with data from redundant source or could be checked against known ranges within which results should be.

How a system actually deals with a fault is a broad subject, however most solutions make use of some form of redundancy: hardware, information, time, or software. [1]

Passive hardware redundancy implies that a system is able to mask a fault without reconfiguration, redundant hardware operates in parallel performing the same task, and a voter decides which output to take. A Triple-Modular Redundancy (TMR) system is a passive hardware redundancy implementation. Active hardware redundancy necessitates that the system must be able to detect the presence of a fault and dynamically reconfigure in order to take advantage of redundant components. An example is a system where checkpoints are taken periodically and tasks running a particular node may be relocated and restarted on another (spare) node if the original node becomes faulty.

Information redundancy most often refers to the use of data-level codes. Here a unit of data is encoded, generally in such a way as to compress it, and the encoded information may be used to check or even correct the integrity of the original data at a later time. Parity bits, m-out-of-n codes, and checksums are examples of codes that allow error detection, while Hamming and other codes may be used to both detect and correct errors.

Time redundancy makes use of a backup, or recovery, plan that is put into effect upon detection of a fault in the primary. That is, a backup, sometimes called ghost, process could be started if the primary task were detected to have failed.

Software redundancy refers to techniques such as n-version programming, or the recovery block approach. Here two or more, perhaps different, versions of a piece of software are provided so that if one fails due to one fault, another version will hopefully be unaffected by that same fault. The recovery block approach is an example of both time and software redundancy - the recovery block is a program that may take less time to execute, perhaps using a different algorithm, than the primary, and is not scheduled unless the primary will miss its deadline.

1.3 Levels of Fault Tolerance

Thus we can have redundancy at different levels of a system. However, in this paper we apply an additional classification of fault tolerant techniques: we deal with fault tolerance at two levels, system-level and application-level.

- *System-Level Fault Tolerance:* This encompasses redundant system components and recovery actions taken by the system. The components involved might include operating systems, scheduling/allocation algorithms, redundant hardware/network configurations, and recovery algorithms. For example, in the event of a failed processing unit, the component of the system responsible for fault tolerance would take care of rescheduling the task(s) which had been executing on the faulty node, and restarting them on a good node from the previous checkpoint.

- *Application-Level Fault Tolerance:* Application-level fault tolerance encompasses redundancy and recovery actions within the application software. Here various tasks of the application may communicate in order to learn of faults and then provide recovery services, making use of some redundancy.

It should be noted that these classes of fault tolerance may each be implemented using any of the four types of redundancy. The classification we propose here is independent of the redundancy techniques used at each level.

If we consider system-level fault tolerance alone, we have found that in certain situations it alone does not suffice to prevent a deadline miss upon the failure of a node. Our application-level solution involves a degree of redundancy within each of the parallel application processes, and the application as a whole is able to make use of that redundancy in the event of a fault to ensure that the required level of reliability is achieved. We consider parallel, real-time systems, and how fault tolerance may be imparted to such applications through Application-Level Fault Tolerance.

Chapter 2 introduces our technique, then in Chapter 3 we present other work that is closely related, and compare our technique to others that have been studied. In Chapter 4 we present the benchmark applications that we have integrated our technique with thus far, and present our results. Chapter 5 analyzes Application-Level Fault Tolerance and classifies applications for which ALFT is useful and describes how ALFT can be implemented with these applications. Chapter 6 described future research directions which could be embarked upon to further ALFT research, and Chapter 7 concludes the thesis.

CHAPTER 2

APPLICATION-LEVEL FAULT TOLERANCE

Application-Level fault tolerance is a straight-forward technique that may be applied to many data parallel applications. That is, applications where there are multiple processes, each carrying out similar computations, but on different sets of data, or on unique parts of the same dataset. The technique uses redundancy in the form of extra work done by each parallel process of the application. Each process takes, in addition to its own distinct work load, some portion of its *neighbor*'s work load, as depicted in Figure 3. Each process then carries out both its own, original, work and overlaps part of its neighbor's, but makes use of the redundant information only in case this neighbor becomes faulty. Thus if one of the processes is found to be faulty, the results that its neighbor has computed may be used in final output in place of those from the faulty node/process. In addition, when the faulty process is replaced or restarted it can inherit an up-to-date copy of any necessary state information from its neighbor. We now provide a more thorough description of the technique, including the division of load and how faults are masked.

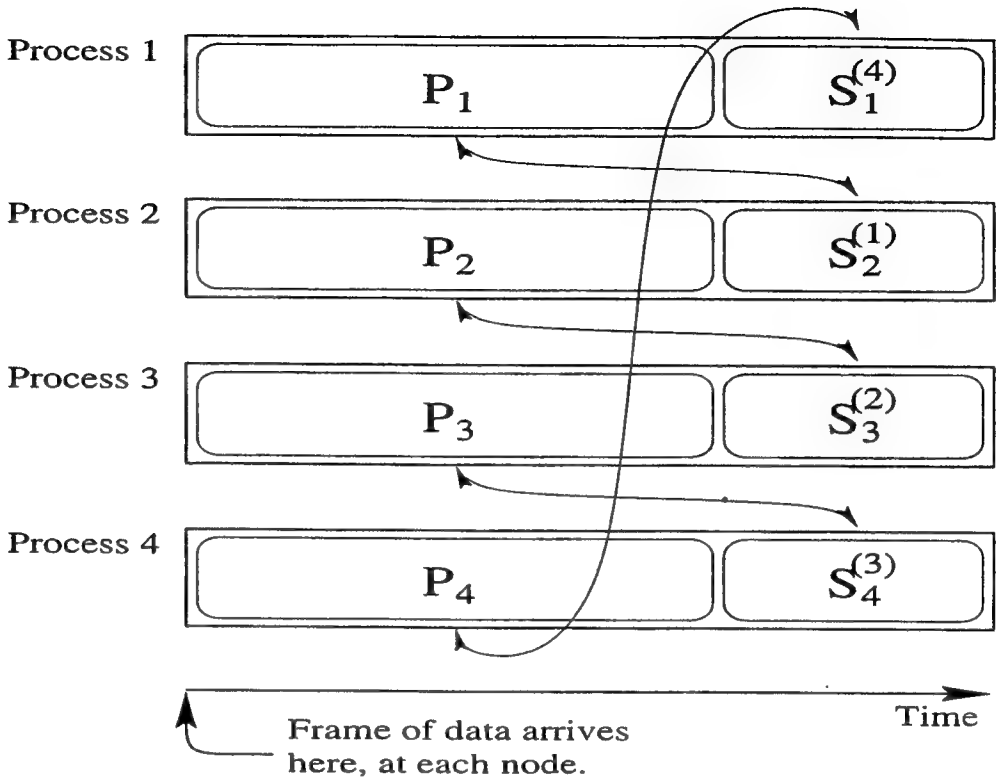


Figure 1: Architecture of a data-parallel application with application-level fault tolerance.

2.1 Division of Load

The extent of duplication between two neighboring nodes will greatly affect the level of reliability which can be achieved. Duplication arises from the way we divide the data set among the parallel processing nodes. First, each frame of data is divided as evenly as possible among the nodes. The section of the process that takes on this set of data is the primary task section, P_i. Then we assign each node, n_i, some additional work: part of its neighbor, n_{i-1}'s, primary task. The section of the process that takes on this set of data is the secondary task section, S_i. In other words,

- The *primary task section*, P_i, refers to the calculations which node n_i carries as part of the original application.

- The *secondary task section*, S_i, refers to the calculations which node n_i carries out as a backup for its neighbor, n_{i-1}. Node n_1 is the backup for the highest numbered node. The secondary section S_i will be kept in synchronization with the primary P_{i-1}.

2.2 Detection of Faults

There are two ways in which fault detection information can reach the various application processes. In the first, the system informs the application of a faulty node, and the second is through specific timeouts at the phase of the application where communication is expected. The former would typically incur the cost of periodic polling, while the latter could result in late detection of the fault. Although the exact integration of application-level fault tolerance would vary depending on the fault detection technique chosen, the effectiveness of our technique should not.

2.3 Fault Recovery

If, at a deadline prior to that of the frame, node n_i is discovered to be faulty and is unable to output any results, then node n_{i+1} which is serving as its backup will send as output S_{i+1}'s data in place of the data that n_i is unable to supply. In the meantime, the system will be working on replacing or restarting the process

that was interrupted by the fault. In fact, the system's job here is made easier by the fact that if the process has to be restarted on another node, the process data segment no longer needs to be moved. When the process is rescheduled, it will make use of the information maintained by its secondary on its behalf in order to pick up where it left off before the fault. This way, the application fault tolerance is able to work in conjunction with the system fault tolerance. This will help even in the case of transient faults, in that the application-level fault tolerance allows more leeway to postpone the restarting of the process on another node, in the hope that the fault will disappear.

2.4 Extension to a higher level of redundancy

Our technique guarantees the required reliability in the presence of one fault but could also withstand two or more simultaneous failures depending on which nodes are hit by the faults. For example, if the nodes running processes 1, 3, and 5 fail, on a six-node system, the technique would still be able to achieve the required reliability. Of course, this is contingent on the assumption that the processes on the faulty nodes are transferred to a safe node and restarted by the beginning of the next frame.

CHAPTER 3

RELATED WORK

From the description above, it is obvious that our technique shares some similarities with the recovery block approach, and n-modular redundancy, and Algorithm-Based Fault Tolerance. The following sub-sections introduce the reader to the latter brand of application fault tolerance, and then we compare and contrast our technique with each of the three.

3.1 Algorithm-Based Fault Tolerance

3.1.1 An Introduction

Algorithm-Based Fault Tolerance is a technique developed by J. A. Abraham and K. Huang in 1984 [3]. The technique focuses on matrix operations, and how matrices and matrix operations may be made more fault tolerant by use of information redundancy. First the information contained in a matrix is encoded and incorporated into a new matrix. Then the matrix operations are redesigned so as to be able to operate on the matrix containing both encoded and unencoded data. Further, the output of these operations is a matrix that is encoded in the same manner as the inputs. The integrity of the resulting data may then be checked for

errors, and errors could be corrected depending on the encoding used. Much mathematical and theoretical work has been carried out in regard to algorithm-based fault tolerance. We will next further introduce the technique and summarize this work.

A basic, and often used example of Algorithm-Based Fault Tolerance is this: Suppose we have a general matrix multiplication operation, $A \times B = C$, and want to introduce fault tolerance into the operation itself. First take the input matrices and perform checksums across the rows of one (A_r), and checksums down the columns of the other (B_c). Next notice that without altering the original matrix multiplication routine: $A_r \times B_c = C_{rc}$. That is, when we multiply a row checksummed matrix by a column checksummed matrix, the resultant matrix contains row and column checksums that are correct for the resulting data. The resulting checksums may then be used to test the integrity of the data within the matrix C_{rc}, allowing a certain number of errors to be detected. (Naturally not more than one error per row/column.)

As presented, this technique increases the size of matrices in the system by 1 in each dimension, thus necessitating use of more processors to handle the checksum elements. In addition, the system designer might want the checking data stored in and operated upon in hardware elements that are distinct from the ones containing the original data, so that a fault in one won't affect both the original and checking data.

3.1.2 Continued Work

From its conception in 1984 to the present time, there has been a slow, but steady stream of work on the topic. Key areas of work have involved searching for bounds on the computational overhead required by the technique, and on the use of new and different error correcting or detecting codes in conjunction with matrix operations.

Abraham continued development of the technique, defining bounds on the time and processing overhead required for the technique [4]. F. Luk, and others, at Cornell, carried out further mathematical analysis of the technique during the late 80's [5]. D. Rosencrantz and S. Ravi continued in the early 90's, searching for improved bounds on various overheads associated with the technique, and found ways to measure the performance of the algorithm-based technique [6] [7].

More recently, Tao, Hartmann, and Han [8] have published what they term "partial checksum" techniques. These include a Lengthened Hamming Code (LHC) method, and a Single Error Correcting/Double Error Detecting (SEC/DED) code. In addition, Yajnik and Jha [9] describe a system that goes a bit beyond the traditional Algorithm-Based fault tolerance. They attempt to solve the problem of what to do once an error is detected and corrected. Detection of an error can give the system some information as to which part of the system might contain the fault. With this information, the system can be designed so that workload can be shifted from the faulty node to another node if available. Their technique naturally strives to make this configuration change as gracefully as possible in order to disrupt the system as little as possible.

3.1.3 How ALFT Compares

Algorithm-Based Fault Tolerance is a technique whose primary function is to both detect and correct errors at the word or data-structure level. Specifically, algorithm-based techniques have been geared toward providing fault tolerance in matrix operations in multiprocessor systems. Although there are many systems (scientific, navigational, or other) which consist of a large proportion of matrix operations, there are many other similar applications that have been developed which either contain fewer matrix operations (and more scalar operations) or consist of literally smaller matrices. To the end of creating a fault tolerant system, if an application has a good proportion of scalar operations, traditional Algorithm-Based Fault Tolerance might not be the answer. In addition there is a case to be made against using the traditional technique if matrices are smaller in size, 3×3 or 4×4 for example.

In the case of smaller matrices, the traditional Algorithm-based fault tolerance will require a substantial proportion of either redundant hardware or computational overhead. If one takes a 3×3 matrix, and adds to it a row and column check sum the number of matrix elements that must be operated upon increases from 9 to 16. These additional elements could mean nearly 50% more work for the system!

Our technique is better suited to provide fault tolerance for parallel distributed systems for which traditional Algorithm-Based Fault Tolerance is inappropriate. That is, applications which are parallel, perhaps real-time, and either don't contains significant proportion of matrix operations or where smaller matrices are more

common. We have located two such applications and describe them in greater detail in the next section.

3.2 Recovery Block Approach

The recovery block approach combines elements of checkpointing and backup alternatives to provide recovery from hard failures [10]. All tasks are replicated but only a single copy of each task is active at any time. If the active copy of a task fails, the backup is executed. Generally backup copies make use of an algorithm fundamentally different from that of the primary so as to both allow for a quick, imprecise result to be output and to guard against common-mode failures. The backup task may be started from the beginning of the computation (which increases the chances of a deadline miss) or else executed from the most recent checkpoint [?]. The later option requires that the active copy of the task periodically copy (checkpoint) its state to its backups. This can entail a large amount of overhead, especially when the state information to be transferred is large. Such is the case with the applications that we are dealing with. In addition, the process of recovering from the fault is entirely in the hands of the system when using the recovery block approach.

In addition, when using the recovery block approach, the system would have to be more intricately aware of the needs of the application in order to be able to start the appropriate ghost-copy of the task that has failed (in whatever way). Application-Level Fault Tolerance on the other hand imposes only very limited

additional load or complexity upon the system or its scheduler, even in the face of a fault. Our technique requires that the system knows very little about the specifics of the application. This helps define a boundary between the system-level recovery techniques and application-level recovery techniques, and simplifies the design of both.

3.3 N-Modular Redundancy

N-Modular Redundancy is a well-known fault tolerance technique. A number of identical copies of the application are run on separate machines, the output from all of them is compared, and the majority decision is used [2]. This technique however, involves a large amount of redundancy and is thus costly.

Our technique does in fact provide fault tolerance similar to that which two complete physical systems would, in the presence of one faulty node. That is, both techniques guarantee the they will remain operational in the face of a single fault. Our technique, however, achieves this with much less hardware, but does require fault detection and location information and a small amount of computational redundancy on the existing hardware.

CHAPTER 4

INTEGRATION WITH TWO APPLICATIONS

We have integrated and thoroughly tested our technique with two parallel, real-time benchmark applications. Each of the benchmarks has the form shown in Figure 2. They consist of multiple, parallel *application* processes, which are fed with input data from a source - in this case, a *source* process which simulates a radar system or an array of sonar sensors. When the parallel computations are complete, the results are output to a *sink* process, simulating system display or actuators. The fault-tolerance technique is concerned with the ability to withstand faults at the parallel processes/nodes. We will discuss both benchmark applications briefly.

4.1 The Benchmarks

4.1.1 The RTHT Target Tracking Benchmark

The Honeywell Real-Time Multi-Hypothesis Tracking (RTHT) Benchmark [11, 12], is a general-purpose, parallel, target-tracking benchmark. The purpose of this benchmark is to track a number of objects moving about in a 2-dimensional coordinate plane, using data from a radar system. The data is noisy, consisting

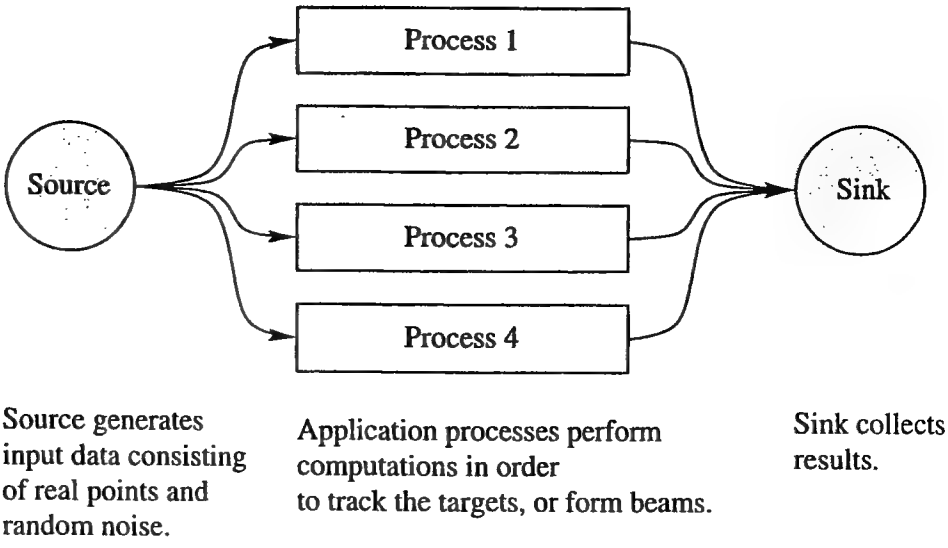


Figure 2: Software architecture of both the RTHT and ABF benchmarks

of false targets and clutter, along with the real targets. The original, non-fault-tolerant application consists of two or more processes running in parallel, each working on a distinct subset of the data from the radar. Periodically, frames of data arrive from the radar, or source process in this case, and are split between the processes for computation of hypotheses. Each possible track has an associated hypothesis which includes a figure of likelihood, representing how likely it is to be a real track. A history of the data points and a covariance matrix are used in generating up-to-date likelihood values.

For every frame of radar data, each parallel process performs the following steps: 1) Creation of new hypotheses for each new data point it receives, 2) Extension of existing hypotheses, making use of the new radar data and the existing covariance matrix, 3) Participation in system-wide compilation or ranking of hypotheses, led by a *Root* application process, and 4) Merging of its own list of hypotheses with

the system-wide list that resulted from the compilation step. The deadline of one frame's calculations is the arrival of the next frame.

By evaluating the performance of the original, non-fault-tolerant, benchmark when run in conjunction with our RAPIDS real-time system simulator [14], it became apparent that despite the inherent system-level fault tolerance in the simulated system, the benchmark still saw a drastic degradation of tracking accuracy as the result of even a single faulty node. Even if the benchmark task were successfully reassigned to a good node after the fault, the chances that it had already missed a deadline were high. This was in part due to the overheads associated both with moving the large process checkpoint over the network and with restarting such a large process. Once the process had missed the deadline, it was unable to take part in the compilation phase and was forced to start all over again and begin building its hypotheses anew. This took time, and caused a temporary loss of tracking reliability of up to five frames. Although better than a non-fault-tolerant system, in which that process would simply have been lost, it was not as reliable as desired.

Two points need to be addressed, in order to improve the performance of this benchmark in the presence of faults: 1) The overhead involved with moving such a large checkpoint and 2) A source of hypotheses for the process to start with after the task's restart.

Our measure of reliability with respect to this application is the number of *real targets* successfully tracked by the application (within a sufficient degree of accuracy) as a fraction of the exact number of real targets that should have been

tracked. To simplify this calculation, the number of targets is kept constant and no targets enter or leave the system during the simulation.

4.1.2 The ABF Beam Forming Benchmark

The Adaptive Beam Forming (ABF) Benchmark [13] is a simulation of the real-time process by which a submarine sonar system interprets the periodic data received from a linear array of sensors. Similarly, it has been developed at Honeywell. The goal is to distinguish signals from noise and to precisely identify the direction from which a signal is arriving, across a specified range of frequencies. In this implementation, the application receives periodic samples of data as if from the linear sensor array. The data is generated so that it contains four reference *beams*, or signals, arriving from distinct locations in a 180-degree field of view, along with random noise.

The application itself consists of several application processes, each attempting to locate beams at a distinct subset of the specified frequency range. Frames of data for each frequency are "scattered" periodically from the source process. Output, in the form of one beam pattern per frequency, is "gathered" by the sink process. Figure 2 depicts a typical beam pattern output, shown here at frame 18, frequency 250Hz, with reference beams at -20, -60, 20 and 60 degrees.

In order to detect and locate beams, each application process performs calculations according to the following loop of pseudo-code, for each frame of input.

```
for_each ( frequency ) {
  Update dynamic weights.
  for_each ( direction of arrival) {
    Search for signal, blocking out interference
```

```
        from other directions and frequencies.
    }
}
```

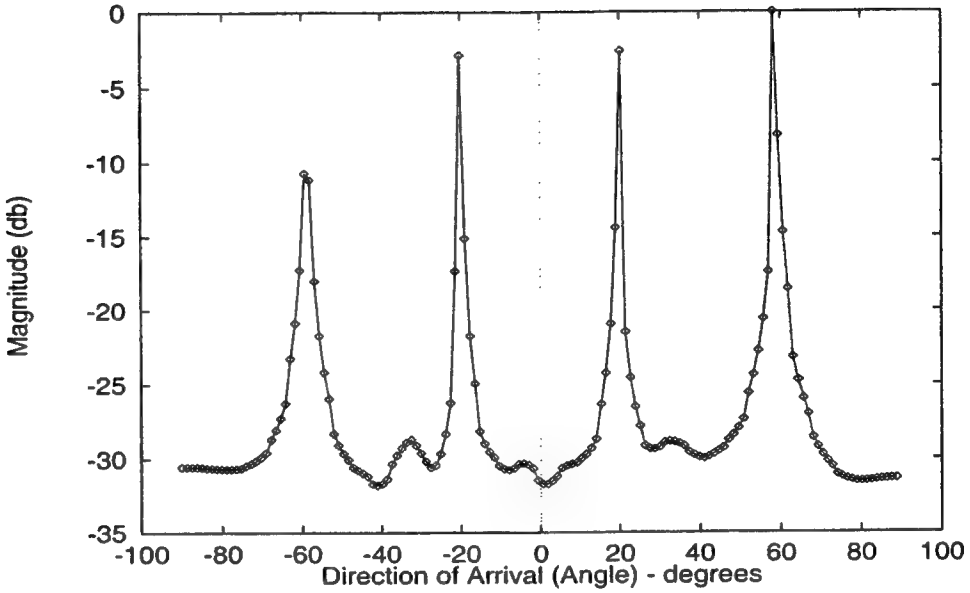


Figure 3: Typical beam pattern output.

That is, for each frequency, the process first updates a set of weights that are dynamically modified from frame to frame. Applying these weights to the input samples has the effect of forming a beam which emphasizes the sound arriving from each direction. The process searches in each possible direction (-90 to 90 degrees) for incoming signals. The granularity of this direction is directly related to the number of sensors.

In addition, at the start of a run, there is an initialization period in which the weights are set to some initial values, and then 15 to 20 frames are necessary to "learn" precisely where the beams are.

It is evident that this sort of application faces reliability problems similar to those of the RTHT benchmark. If a processing element fails, all output for those frequencies is lost during the down time, and when the lost task is finally replaced by the system, it will have to go through a startup period all over again. Here too, the data segments of these processes are very large, creating a considerable overhead if checkpointing is employed. To avoid the delay associated with this overhead and be able to maintain full output during the fault and quick restart after the fault, application-level fault tolerance must be employed.

The quality of the ABF output is measured with two tests applied to the resultant beam pattern. In the Placement Test we check whether the direction of arrival of the beam has been detected to within a certain tolerance. In the Width Test the aim is to determine how accurately the beam has been detected by measuring the width of the beam, in degrees, at 3db down from the peak. A beam that passes both tests is considered to be correctly detected.

4.2 Benchmark Integration Specifics

Here we discuss specific details regarding the integration of our technique with each of the benchmarks.

4.2.1 Integration in the RTHT benchmark

In the RTHT Benchmark, the data element that is operated on in parallel is the hypothesis. That is, each secondary task section creates and extends some fraction of the total number of hypotheses created and extended by the process

for which it is secondary. The amount of secondary redundancy is expressed as a percentage of the number of hypotheses extended by the primary.

Redundancy is implemented in the following way: At the beginning of each frame, the source process broadcasts the input radar data, and hypotheses are created and extended as before, except that additionally the secondary extends a percentage of those extended by the corresponding primary. The secondary section S_i is kept in synchronization with the primary P_{i-1} via the compilation process, which in this case is again a process-level broadcast communication, so that no extra communication is necessary. If, at compilation time, node n_i is discovered to be faulty and is unable to participate in that frame's compilation, then node n_{i+1} which is serving as its backup will make use of S_{i+1}'s data in the compilation process in place of the data that n_i is unable to supply.

When the process is rescheduled, it will make use of the hypotheses extended by its secondary on its behalf in order to pick up where it left off before the fault. This information is obtained from the secondary process by way of the compilation process - the newly rescheduled process merely listens in on the compilation process and copies those hypotheses which have been extended by its secondary.

4.2.2 Integration in the ABF benchmark

There are two ways in which we have integrated application-level fault tolerance with the ABF Benchmark. They differ in the manner in which the secondary abbreviates the calculations of the primary so as to obtain a full set of results. The methods are:

- The Limited Field of View (Limited FOV) Method in which the secondary looks for beams at every frequency as in the primary, however it searches only a subsection of the primary's field of view (divided into one or more segments). Ideally the secondary will place these "windows" at directions in which beams are known to be arriving. We impose a minimum width of these windows, due to the fact that if an individual window is too narrow, the output could always (perhaps erroneously) pass the width-based quality test, described above (at the end of section 2). The amount of redundancy is expressed as the percentage of the field of view searched by the secondary.

- The Reduced Directional Granularity Method in which the secondary looks for beams at every frequency and in every direction, but with a reduced granularity of direction. The amount of redundancy is expressed as a percentage of the original granularity computed by the primary.

Both techniques serve to reduce the computational time of the secondary task set, while maintaining useful system output. In addition, the two techniques may be employed concurrently in order to further reduce the computational time required by the secondary task.

To implement either variation of the technique, the input frame of data is scattered a second time from the source to the application processes. This is time rotated, so that each process receives the input data of the process for which it is a secondary. Each process first carries out its primary computational tasks, and then carries out its secondary task. At the frame's deadline, if a process is detected

to be down, the sink will "gather" output from the non-faulty processes, including the backup results from the process

that is secondary for the one that is faulty. In the event of an application process being restarted after a fault, it will receive the current set of weights from its secondary in order to jump-start its calculations.

Some synchronization between primary and secondary is required in the Limited FOV Method. It is a small, periodic communication in which either the sink process or the primary itself tells the secondary at what frequencies and directions it is detecting beams. Such synchronization is not necessary for the Reduced Granularity Method.

4.3 Complete Results from RTHT and ABF Benchmarks

4.3.1 The RTHT Benchmark

When applied to the RTHT benchmark, we found that only a small amount of redundancy between the primary and secondary sections is necessary in order to provide a considerable amount of fault tolerance. Furthermore, the increase in system resource requirements, even after including overheads of the technique's implementation, is minimal compared to that of other techniques, in achieving the same amount of reliability. These points are demonstrated in Figures 4, 5, and 6. Each run contains 30 targets which remain in the system until the end of the simulation (the 30th frame), as well as some number of false alarms. The case

when only system-level fault tolerance exists corresponds to the case when the secondary extends 0% of the primary hypotheses.

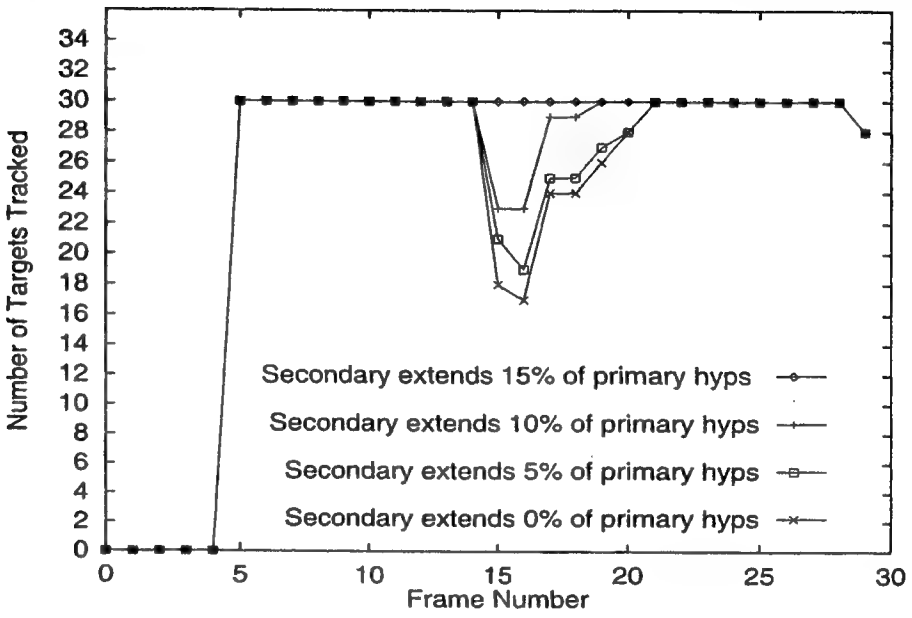


Figure 4: Tracking accuracy, in number of real targets tracked for a given percentage of redundancy.

In Figure 4 we see the number of targets which are successfully tracked, when we have just two application processes and a fault occurs at frame 15. (In this case there were roughly 80 false alarms per frame of data.) In this run, 15% redundancy allows us to track all of the real targets, despite the fault. We can attribute the fact that a small amount of redundancy can have a great effect on the tracking stability, to the fact that the hypotheses which are being extended by the secondary are the ones *most likely* to be real targets. At the beginning of the compilation phase, each application process sorts its hypotheses, placing the *most likely* at the head of the list for compilation. Thus, at the beginning of the next frame, each

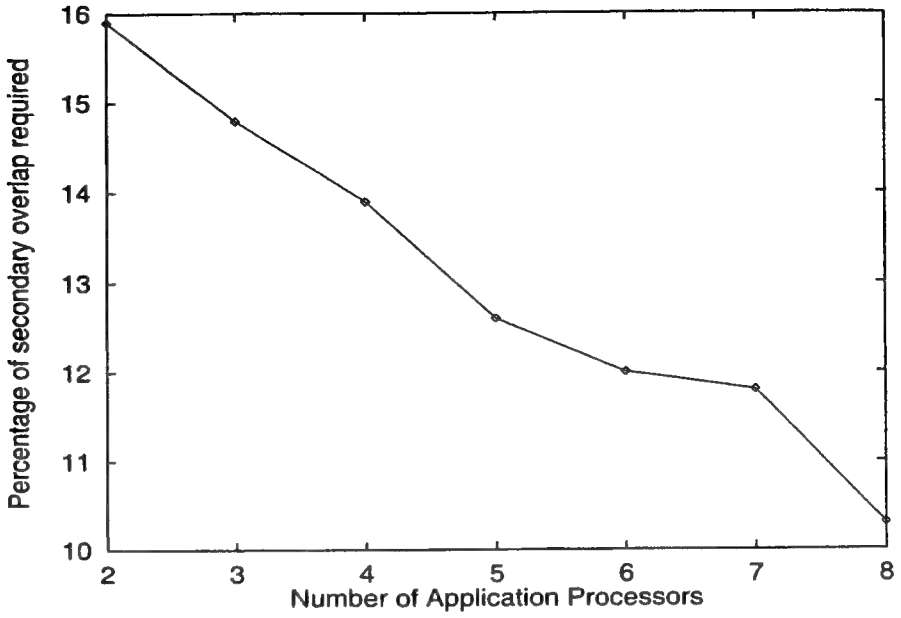


Figure 5: Average minimum percentage of secondary overlap required to miss no targets despite a fault at one node.

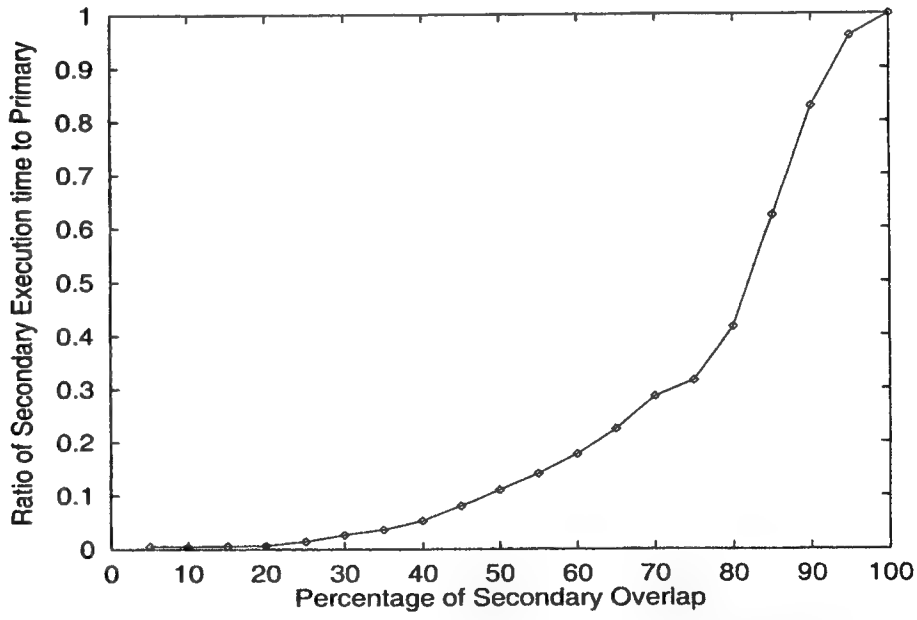


Figure 6: Ratio of time taken to compute the secondary hypothesis to the time to compute the primary hypothesis versus the percentage of secondary overlap.

application process and its secondary begin extending those hypotheses with the highest chance of being real targets.

To refine this point, Figure 5 shows the average percentage of redundancy required for a given number of application processors and a single fault, as before. The amount required shows a gradual decrease as we add more processors. We can attribute this to the fact that the chances of a single process containing a high percentage of the real targets decreases as processors are added.

In addition, a proportionately small load is imposed on the processor by the computation of the secondary task set, as seen in Figure 6. This can be attributed to the fact that the extension of a hypothesis whose position and velocity are known precisely, does not take as much time as extending to those hypotheses which are less well-known. And since the *most likely* hypotheses are generally the most well-known *and* are the hypotheses which the secondary extends, the amount of processor time taken to execute the secondary task is proportionally much smaller.

4.3.2 ABF Benchmark Results

When we integrate application-level fault tolerance with the ABF benchmark, we find that only a small amount of redundancy is necessary to ensure complete masking of single frame faults. With either variation (reduced granularity or limited FOV method) we see that a secondary redundancy of 33% is adequate to provide complete and accurate results in the faulty frame, and the following frames (after the faulty process is restarted). If we combine the two techniques, we see

an even further reduction in the computational effort imposed by the secondary in order to mask the fault. We have not taken additional network overhead and/or latency into account in figures of overhead - they refer solely to computational overhead. Network overhead will depend greatly on the medium used. In particular, a shared medium would allow the secondary to "snoop" on the primary's input and output, eliminating the need for additional communication.

All results were obtained by running simulations with 75 sensors and four reference input beams for 50 frames. There are two application processes, and a fault occurs in one of them at frame 30. Results are presented and discussed for three redundancy methods: the Limited FOV method, the Reduced Granularity method and a Combined method (a combination of the first two). The quality of results is assessed by totalling the number of beams that were tracked successfully (or by totalling the number of beams not tracked correctly). Here, there are four input beams at each frequency, and 32 frequencies - making 128 beams in all. As an example, Figure 7 presents the results for several runs of the ABF benchmark while utilizing the Limited FOV redundancy method alone, with a single processor fault occurs at frame 30, lasting one frame. We see that a 30% overlap is adequate to preserve all beam information within the system despite the loss of one process in frame 30. We have tabulated the results for all three methods in Table 1.

ABF Results: Limited FOV alone

As we see in Table 1, roughly 30% secondary overlap is adequate to provide full masking of the fault. The computational overhead imposed by the secondary

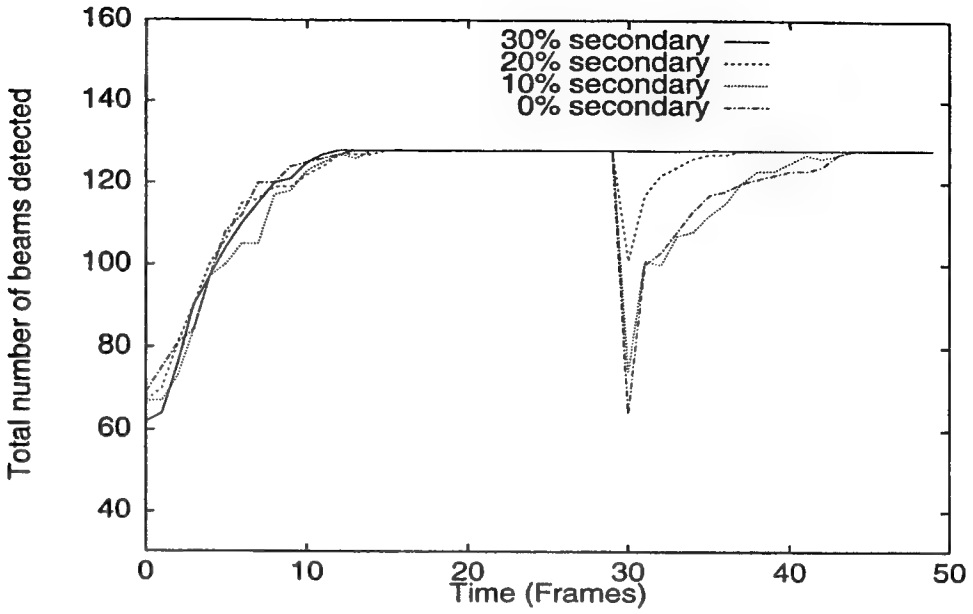


Figure 7: The number of beams correctly tracked in each frame, for the given levels of redundancy, for the Limited Field of View Method. A single process experiences a fault of duration one frame, at frame 30.

is about 30%. In addition, Figure 8 shows the rather linear increase in overhead as we increase the fraction of overlap.

Associated with this technique however, is a potential dependence on the number of beams detected in the system, as described earlier. In order to ensure that the width test, applied to the output, can fail, we impose a minimum window-width. This minimum width dictates that for a given amount of overlap, there is

Redundancy Technique	Secondary Overlap	Computational Overhead
Reduced Granularity	33%	35%
Limited FOV	30%	30%
Combined - 30%FOV,50%Granularity	15%	17%

Table 1: Amount of secondary overhead imposed by various redundancy methods, each of which is capable of fully masking a single fault.

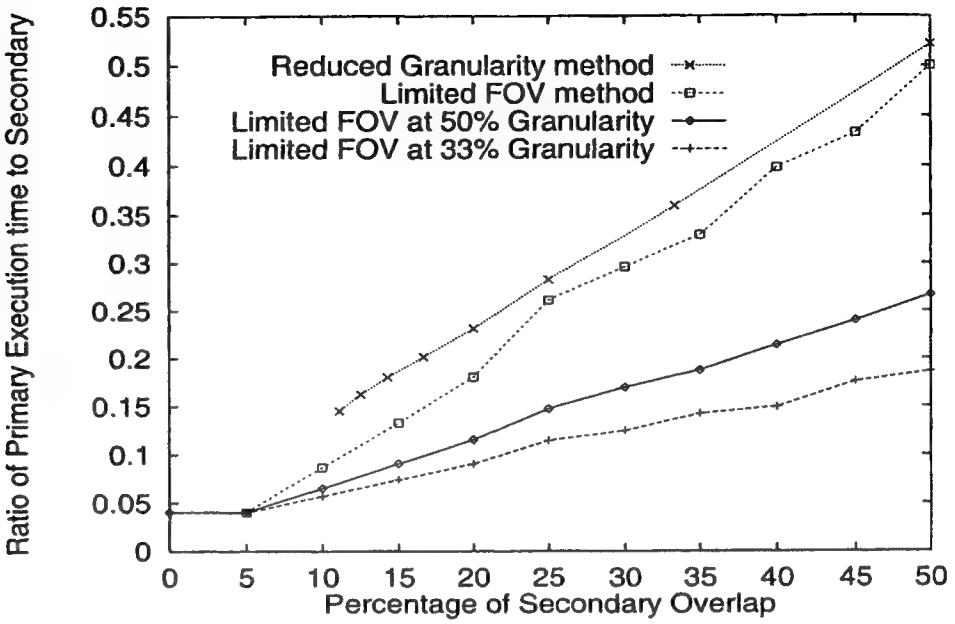


Figure 8: The ratio of secondary to primary execution time for the variations of application-level fault tolerance integrated with the ABF Benchmark versus the percentage of secondary field of view overlap.

a maximum number of windows in which the secondary may search for beams. If there are more beams than the maximum number of windows then some may be missed by the secondary search, depending on the direction of arrival. Here, the system or application will need to decide, based on either dynamic feedback or on some static rules, which beams are most important for the secondary to track.

ABF Results: Reduced Granularity alone

Here, too, we see that, according to Table 1, operating the secondary at 33% of the granularity of the primary results in complete masking of the fault, and that this imposes a 35% overhead to the processing node. Figure 8 again shows a linear relationship between the computational overhead and overlap, and indicates that the overhead of the method itself is a bit higher than that of the Limited

FOV method. When considering the Reduced Granularity method, we see no dependence on the number of beams detected, although beams could be missed if their peaks were within a few degrees of each other, and the granularity were very coarse. This was not a factor here, as our four reference beams were evenly placed across the 180 degree field of view.

ABF Results: Combined methods

When we combine these two techniques, we see the greatest reduction in computational overhead of the secondary task. As shown in Table 1, a 30% field of view combined with a 50% granularity maintains the tracking ability similar to that of either one alone, yet cuts the computational overhead nearly in half. This reduction is illustrated in Figure 8, in the lower two curves, representing the overhead imposed as we vary the field of view and make use of 50% and 33% granularity respectively. Thus we see that both techniques may be employed in parallel in order to see the greatest reduction in secondary computational overhead while still being able to meet the goals of Application-Level Fault Tolerance.

CHAPTER 5

ALFT EXTENDED

There is a need to quantify our technique at a general level, so that the application designer will easily know if an application is well suited for our technique or not. We need to stress that our technique is not a single, fixed set of steps to be followed to achieve fault tolerance in all applications, but a technique to be integrated at the application level where the exact implementation will depend upon the application itself. Using Application-Level Fault Tolerance, the system designer can provide fault tolerance in the way that best suit the application.

In this chapter we will discuss general characteristics that are assumed prior to applying our technique and then indicate how the designer should approach implementing our technique. Discussion of the potential overheads follows is included, along with example applications in order to illustrate the ideas presented.

5.1 What Makes an Application Suitable for ALFT?

We have thus far only considered applying our technique to applications which display certain general characteristics. This is not to say that the ideas we present

are applicable nowhere else, but that we will focus on systems with these characteristics.

We believe that our technique will be most beneficial to applications that are periodic and data-parallel, where parallel task sections are scheduled concurrently, on distinct processors. Often these same applications will be computationally intensive and large, memory wise.

Our fault tolerance is able to take advantage of the parallelism by integrating, on each processor, a small part of a neighboring processor's work. Thus if the neighbor is struck by a fault, some results may be output on its behalf. There must exist some un-utilized time in the period of the application, or there must be some laxity to extend the period of the application, in order to allow the secondary task time in which to execute. In addition, reduced precision results must be acceptable in the short term (i.e., during the duration of, and recovery from, a fault). If full precision is required in each and every frame, then it is obvious that our technique will require close to 100% duplication of the tasks.

5.2 Effectiveness and Overhead

The question may be asked, how should one determine if a particular ALFT implementation is effective? There are many factors that will determine how effective Application-Level Fault Tolerance will be, however the basic question is: How much (or little) work need be done by the secondary in order to provide useful results when the primary is down and help jump-start the primary upon restart.

The answer to this question will certainly vary with application, type of computation done, algorithm used, and requirements of the end-user. Hence we will leave the precise definition of *effective* to the application designer, as that person will have to weigh the cost with the level of fault tolerance required.

In analyzing cost, one needs to consider what percentage of the primary's original dataset is in fact necessary in order to provide a useful short-term result. Then one needs to study how easy or difficult it will be to partition the original data set, such that the *best* subset of data is operated on by the secondary. Partitioning will involve ordering the data set and/or reducing its granularity in some fashion. The goal of the designer is to find the partitioning method that results in the minimum overhead incurred while providing maximum quality of secondary results.

Overhead will be determined by the sum of computational, communicational, and memory requirements introduced into the application. One must consider overheads such as:

- **Computation:** Creation of the partitioned dataset, allowing the secondary resources to complete execution, fault detection, and deciding when to use the secondary or primary results.

- **Communication:** Transfer of the partitioned dataset to the node where the secondary will run, dissemination of fault information, transfer of secondary results for output when necessary, and transfer of synchronization information periodically between primary and secondary or when the primary is restarted.

- **Memory:** Storage of the secondary's dataset and storage of the secondary's results.

This appears to be a long list; however we have shown that Application-Level Fault Tolerance has proven effective in the RTHT and ABF benchmarks. If the designer is careful to consider all of the above, we feel that fault tolerance solutions involving ALFT can be very effective.

One might think of limiting the cost of ALFT by only allowing the secondary task to execute if the primary is faulty. This might be useful in the case where state information or results are globally known by the parallel processes, so that the state of the neighboring primary is known and updated in each frame locally. However if this is not the case, the designer must weight the cost of executing the secondary normally versus the cost of updating the secondary periodically with the primary's results and/or state information. This is not to say that we assume that the secondary, operating on a reduced task set, will be able to stay in perfect synchronization, but that by running the secondary in each iteration it may not need to be synchronized with the primary as often. Running the secondary despite success of the primary can thus help maintain synchronization, without explicit communication.

Additionally, the coverage of the technique could be increased by forcing the secondary to be computed in each period. The designer will need to consider how likely there is to be a fault between the time at which each primary finishes execution and the time at which results are output. We will see in the next section that some applications exhibit an ongoing dependency between parallel processes.

Here, one phase of execution relies on a full set of results from the previous phase. Thus if the secondary were not automatically executed in the preceding phase, an untimely fault could hold up execution of the following phase. The same holds for any application where the output (or updated state information) of one iteration is required prior to the start of the next iteration.

5.3 Inter-Process Data Dependency Models

We have established that we consider parallel, periodic applications with our technique, and that the application designer must take into consideration all sorts of potential overheads when using our technique. In the next two sections we will discuss ways the task of implementing our fault tolerance can be broken down and classified. Here we will consider how varying degrees of inter-parallel-process data dependency will affect our technique.

Beyond simply finding a way to ensure that the secondary will not impose undue load on the system, it is necessary for the designer to study how the secondary task should be integrated with the primary task in order to be able to share the resources of a single computational node. It is most useful to consider what type of data dependency, and thus what communication, exists between the parallel processes of the application. We have identified three categories:

- No Data Dependency - Communication only at the end of each iteration for the purposes of outputting results.

- Periodic Data Dependency - Over one iteration, communication at the end of a phase exists in order to disseminate results calculated in that phase, as necessary.

- Continuous Data Dependency - Inter-process communication potentially at any time during the task period.

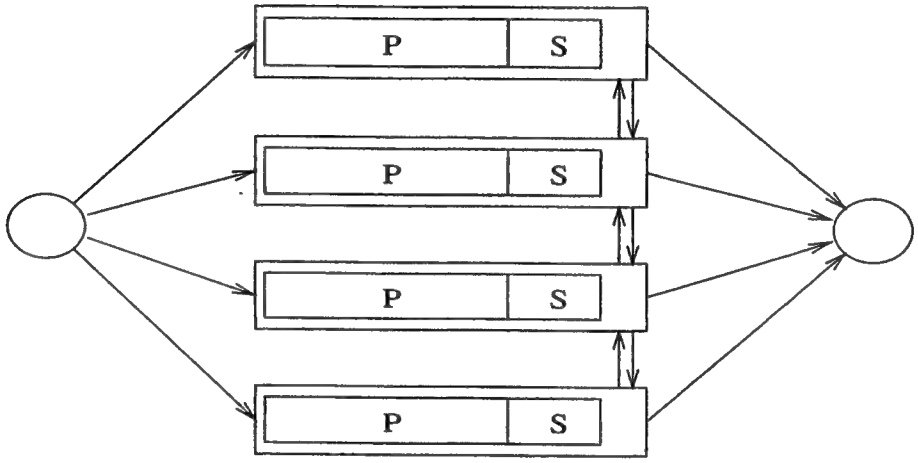


Figure 9: Independent Parallel Process Application with ALFT.

Both the RTHT and ABF Benchmarks fit very nicely into the first category. That is, with both applications, each parallel process carries out its own computation independently of the computations or data of any neighboring processes. In the ABF there is no inter-process communication during computation - results of each process are sent independently to the sink. In the RTHT, there is inter-process communication only after computation has completed, but this communication is for the purposes of organizing the output set of data and does not represent a data dependency between neighboring processes. To implement Application-Level Fault Tolerance in applications of this type, the Primary and Secondary task sections are

distinct. The Secondary task section is computed after the primary, as in Figure 9, and then results are output, with the secondary data made use of in case of a fault.

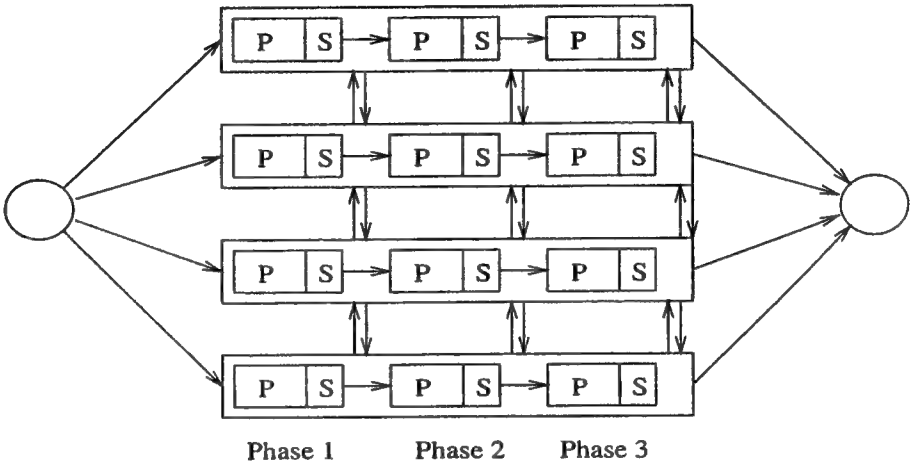


Figure 10: Interleaving of Primary and Secondary in Application with distinct phases of computation and process inter-dependency.

The second type of dependency we see is one in which the system's periodic computation is carried out in distinct phases, with dependency as we will describe. Figure 10 shows an example with three phases; in general there could be any number. In phase two (and perhaps three), each of the parallel processes relies on the set of phase one results. That is, each instance of phase two computation relies on results of one or more other processes' phase one computation as well as its own. A similar dependency might exist between phase two and three. Thus, we see inter-process process communication before or during phase two in order to disseminate phase one results to all processes. In this situation, it would not make sense to block all secondary computation at the end of one period – the secondary computation must be split into phases, just as with the primary computation.

Figure 10 illustrates how the secondary task section is integrated with the original primary task section. The secondary task set is broken into three parts and is interleaved with the primary task sections regarding CPU scheduling. This interleaving is necessary so that, for example, if a node fails between phases one and two, its secondary will be able to take over, and provide phase one results on its behalf during phases two and three. The Particle-In-Cell application, introduced in Section 5.6, is an example of such an application.

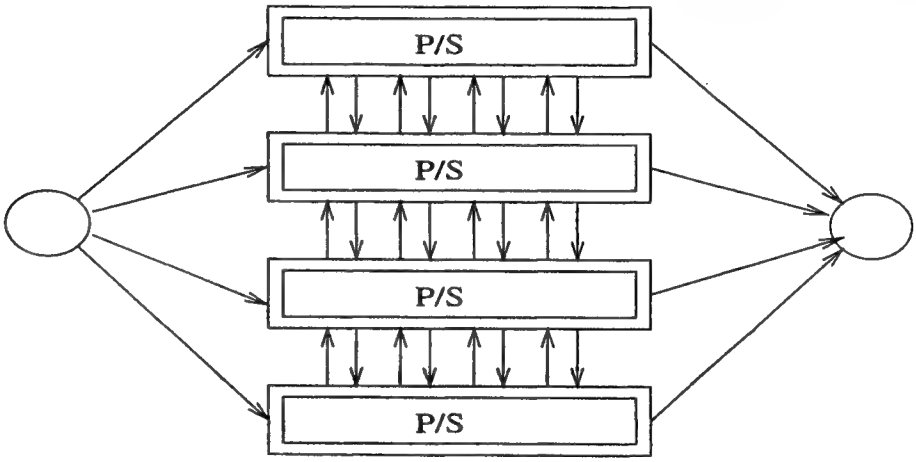


Figure 11: Primary and Secondary task sections tightly integrated in an Application with continuous inter-process dependency.

The third inter-process dependency model is continuous data-dependency. That is, at any time during computation, one process can and will depend on any other process for some information or computation. Thus, many messages may be sent sporadically from process to process during execution. Here it is not feasible to run the secondary task section as a distinct block, even split into phases: At any time, the secondary might be required to output results when the primary is down. Thus, each process must be able to perform primary computations and secondary

computations side-by-side. This tight integration of Primary and Secondary Task sections is illustrated in Figure 11. We provide a 3-d Parallel Image Rendering Application as an example of this type.

5.4 Techniques to Reduce Secondary Runtime

As we have stressed, our technique involves replicating the computations of each parallel process in some reduced form and running this new (secondary) task on a neighboring node. Our technique does not alter the fundamental algorithm of the parallel process. This minimizes changes to the application and algorithm, thereby allowing the primary and secondary to reside within the same processor, making use of the same functions and routines. Thus one of the most important issues to address in order to effectively implement our technique is to minimize the overhead imposed by the secondary task section. We reduce the computational time required by the secondary task, exclusively by reducing the number of elements operated on by the secondary. There are two ways by which we can achieve this:

- Find a way to prioritize these elements and have the secondary only operate on those that are most important.
- Reduce the Granularity of the dataset - if the secondary process is performing computations at discrete data points across a continuous space, then we could reduce the granularity by having the secondary operate, for example, on every other element.

Naturally, the determination of which technique, or combination of the two, is best will depend on the feasibility of implementation and the quality of results that may be achieved. As both are strongly linked to the application itself, we provide a series of questions to help the system designer choose the best technique. We are concerned mainly with how to design the secondary task set such that the computational overhead is minimized and the quality of results is maximized. The first question discusses in what way the application is parallel, the next two determine if it is useful or feasible to reduce the number of elements operated on by the secondary.

1. What dataset(s) are split across parallel computational nodes?
2. Can we reduce the granularity of these dataset(s)?
3. Can we prioritize the elements of the dataset(s) and limit the number of elements operated upon?

If there are multiple parallel data types in the dataset, questions 2 and 3 are asked with regard to each data type.

In the next section, the RTHT and ABF benchmark applications are used to illustrate this question and answer technique, as they have been discussed in detail in Chapter 4. In the last two sections of this chapter we introduce the Particle-In-Cell Simulation and a Parallel 3-d Rendering Application in order to show how our technique can benefit those applications as well.

5.5 Independent Processes: RTHT and ABF Benchmarks

Here we present the RTHT and ABF Benchmark applications in light of the analysis presented in Sections 5.2, 5.3, and 5.4.

1) What data set is parallelized across the nodes?	Hypotheses, representing tracks.
2) Can we intelligently prioritize the Hypotheses?	Yes - Secondary only operates on those most likely to be real targets.
3) Can we reduce the granularity with which the secondary operates on the set of hypotheses?	No - Hypotheses are independent of one other.

Table 2: Implementing ALFT with the RTHT Benchmark.

Table 2 shows how we were able to reduce the computational overhead of the secondary task section in the RTHT benchmark. The set of hypotheses is the most obvious dataset that is parallelized across the system, and is conveniently the best opportunity to limit the overhead imposed by the secondary while maintaining very useful results. This is detailed in Chapter 4. We also observe that the RTHT fits into the "No Data Dependency" communication model. Thus, we chose to integrate the secondary task as a single, distinct section to be executed after the primary is completed.

Table 3 shows how we have generalized the ABF benchmark in the same fashion. Here, there are several facets which are parallelized across the nodes. In our ALFT implementation we have considered only reducing the granularity and number of Field of View quantizations, however one could consider limiting the number of frequencies operated upon by the secondary as well. The ABF is another "No Data

1) What data set is parallelized across the nodes?	Frequencies and Field of View
2a) Can we prioritize the frequencies?	Yes - if we are more interested in one frequency range than another (in the short term).
2b) Can we prioritize the Field of View?	Yes - if we are more interested in one direction than another, (in the short term).
3a) Can we reduce the granularity across the frequency range?	Maybe - if the signals in existence at one frequency are related to or affected by those found at nearby frequencies.
3b) Can we reduce the Field of View granularity?	Yes - a lower quality image will result but might be acceptable in the short-term.

Table 3: Implementing ALFT with the ABF Benchmark.

Dependency" application, so we have integrated the secondary task section as a distinct part of the original primary section, to be executed later in the period.

5.6 Periodic Data Dependency: The Particle-In-Cell Problem

The Particle-In-Cell (PIC) simulation is a simulation of the motion of charged particles within a plasma field. The goal of the application is to predict how the particles move around the area due to electromagnetic forces. The area in question is broken up into a grid of cells, where each cell can contain some number of particles (or none at all) and has some electromagnetic field across it. The application operates on two basic data structures: an array representing the particles in the model (called the Particle Array) and an array of electromagnetic field values at each cell (called the Mesh Grid Array). Each array is updated in each iteration,

representing a time step, δt. Each iteration is composed of four computational steps, as laid out in [15]:

- 1) Scatter Phase - For each particle, calculate the effect of the particle towards net electromagnetic fields at the corner of the cell where it resides.
- 2) Field Solve Phase - Solve Maxwell's equations across the Grid - find the electromagnetic field at each cell taking into account the effects of neighboring cell fields.
- 3) Gather Phase - Use resulting field values to calculate the field at each particle.
- 4) Push Phase - Once we know the field at each particle we determine the net force on the particle and determine its position for the next iteration.

Table 4 discusses how overhead could be minimized when ALFT is implemented with the Particle-In-Cell application.

In order to reduce the computation time of the secondary task, we suggest that during the Scatter Phase, the secondary could operate on a reduced number of particles (Table 4, 2a.), and that during the Field Solve Phase it could operate at a reduced granularity (Table 4, 3b.). In case a processor is detected to be faulty, its secondary can assist the system by both updating the Mesh Grid array, and updating the position of all particles assigned to the faulty processor. When the fault disappears, the secondary could communicate the resulting approximated current state of each particle to the primary in order to partially restore the state of the primary.

1) What data set(s) are parallelized across the nodes?	Elements of both Particle Array and Mesh Grid Array.
2a) Can we prioritize the elements in the particle array?	Yes - when updating the Mesh Grid Array, the secondary could ignore the weaker particles and only account for the stronger ones.
2b) Can we prioritize the elements in the mesh array?	Maybe - if we were more interested in one area of the physical space than another area.
3a) Can we reduce the granularity with which the secondary operates on the particle-array?	No - a particle is completely independent of its neighbors, and should be updated in each frame.
3b) Can we reduce the granularity with which the secondary operates on the mesh-array?	Yes - In updating the Mesh-array, the secondary could perform calculations across several mesh-cells at once, and copying the resulting field values to each of the aggregated cells.

Table 4: Implementing ALFT with the Particle-In-Cell application.

Further, we see that the PIC computation fits the dependency model in which there are distinct phases of dependency during a period. Thus the secondary computation is interleaved, in distinct blocks, with the primary, as in Figure 10. For example, in order to start the Field Solve phase, each processor relies on Scatter Phase results of other processors. There are two reasons for this: first, knowledge of neighboring field intensities is required to solve for the field across the entire region in the Field Solve Phase, and second, in the Gather Phase, if a processor's subset of the Mesh and Particle Array are not 100% overlapping then that processor will require data from other processors in order to determine the field at each of its particles. (This latter scenario is likely to occur, as most load balancing schemes for such applications don't enforce such overlap, as in [15].)

One method for storing data across the system, presented in [16], involves replicating the Mesh grid array information on each processor. This not only simplifies communication when the Mesh and Particle arrays are not fully overlapped but also simplifies communication if a secondary's results need to be used.

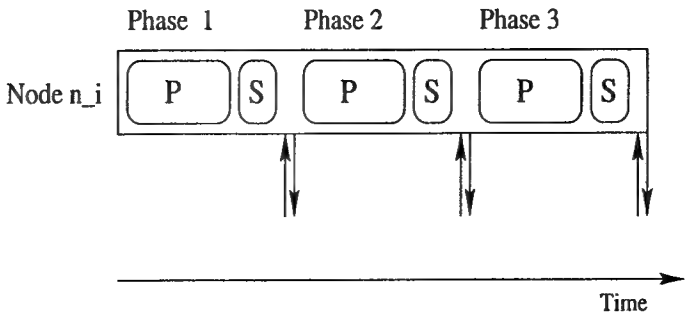


Figure 12: ALFT integrated with one process of the Particle-In-Cell application.

Making use of this inherent data replication, one possible implementation of ALFT might operate as in Figure 12. In phase one, the primary of node n_i first calculates the effect of each of its particles upon the mesh grid cell it occupies, then the secondary does the same computation, but only for those particles that are the most charged (and most likely to affect the overall e-field). Between Phases one and two the parallel tasks communicate in order to make sure they all have the most up to date Mesh field values. Here, if a processor is down, its secondary will fill in for it. In phase two, each primary solves Maxwell's equations for its Mesh grid cells, and then the secondaries do the same, except with reduced granularity, say at every third cell, and copying the computed values to the cells in between. Another similar communication then takes place, to disseminate the updated Mesh Field values. In phase three, each primary computes the Field at each of its particles calculates where the particle will move to in the next iteration. Then the secondary

performs the same computation, on every particle it is (secondarily) responsible for. It is important that the secondary update (even if approximately) the position of each particle so that if a processor fails and is then replaced it can quickly be updated with respect to all particles.

5.7 Continuous Dependency: 3d Image Rendering

The Multi-Pass, Parallel rendering method, presented in [17], is a combination of the Ray-Tracing and Radiosity approaches to image generation. Given a description of the 3d space (and included objects) which is to be displayed, there are two phases to rendering the scene:

- First, a view-independent computation which computes the effects of diffuse light on the scene, using Radiosity techniques.
- Second, a view-dependent computation to compute the effects of specular light at each pixel, making use of ray-tracing techniques.

The object space is divided into small subspaces and these are allocated to the processors. The computing system is a one, two or three dimensional array of processors, each of which are pipelined units, with local memory, specialized for ray processing. Algorithmically, the parallelization of both phases above is very similar. In extremely general terms, both phases consist of packets of data, representing light, that are sent from one part of the scene to another, from processor to processor, just as rays would bounce around a real scene. At each stop, the effect of that surface on the intensity of light at that pixel or surface is computed and added

to the appropriate local memory location (if in the radiosity phase) or the frame buffer (if ray-tracing). Additionally, the object subspaces are allocated to processors such that neighboring subspaces reside on neighboring processors. A single processor thus contains many distinct subspaces, each evenly spaced throughout the scene.

It is fairly obvious from the algorithm used by both phases that there is a continuous data dependency between parallel processes. At any time, a packet could be refracted/reflected from one part of the 3-d scene to any other part of the scene. At that time, there must be a process read and able to handle that packet, be it the process primarily responsible for that part of the scene or the one secondarily responsible (if the primary is down). The secondary computation is tightly integrated with the original primary, in that, when a packet bound for the secondary part of the task arrives at a process, it should be handled immediately. Figure 11. illustrates the tight integration.

Table 5 describes how the Application-Level Fault Tolerance design process might proceed, focusing on reducing the overhead incurred.

As we have seen, data is partitioned across the parallel processes in a number of ways: spatial sub-sections, radiosity "packets", and ray-tracing "packets". Thus there are a number of possible ways to reduce overhead.

One is to reduce the number of subspaces for which the secondary will perform computation. That is, we might be able to prioritize the subspaces such that only the most important subspaces (only those containing important objects) will be taken into account. This however is a relatively crude method.

1) What data set(s) are parallelized across the nodes?	a) Sub-Sections of the Objects space, b) Set of Radiosity Packets, c) Set of Ray-Tracing Packets.
2a) Can we prioritize the subsections of object space?	Perhaps, but seems crude - only operate in places where there are objects of particular interest.
2b) Can we prioritize the ordering of Radiosity Packets?	Yes - Radiosity is calculated with a progressive refinement algorithm, might only send a packet to the secondary if it is within the first few iterations.
2c) Can we prioritize the Ray-tracing Packets?	Yes, a ray-tracing packet might only be forwarded to a secondary task section if it is within the first few computational iterations.
3a) Can we reduce the granularity of Set of Sub-Spaces	No - subspaces are split across processors, can't take two neighboring subspaces and combine them into one.
3b) Can we reduce the granularity of Radiosity Dataset?	Maybe, but reducing the number by prioritized ordering (as above) is probably a better method.
3c) Can we reduce the granularity of Ray-Tracing Dataset?	Maybe, but reducing the number by prioritized ordering (as above) is probably a better method.

Table 5: Implementing ALFT with a 3-d Image Rendering application.

A better way is found by looking at the algorithm followed by both the radiosity and the ray-tracing computations. As radiosity and ray packets are initiated and bounce around the system, the value of light at the particular surface (radiosity) or pixel (ray-tracing) is iteratively updated and refined. In both cases we can make a general assumption that the earliest iterations (or even the first iteration!) will have the greatest effect upon the value at each pixel. In this way, a prioritized ordering exists in both phases. We can specify that a secondary task should not refine the information at a pixel or surface when beyond a specified iteration. Such

a technique could easily be implemented by adding a counter to each ray-tracing or radiosity packet, and incrementing that counter with each hop.

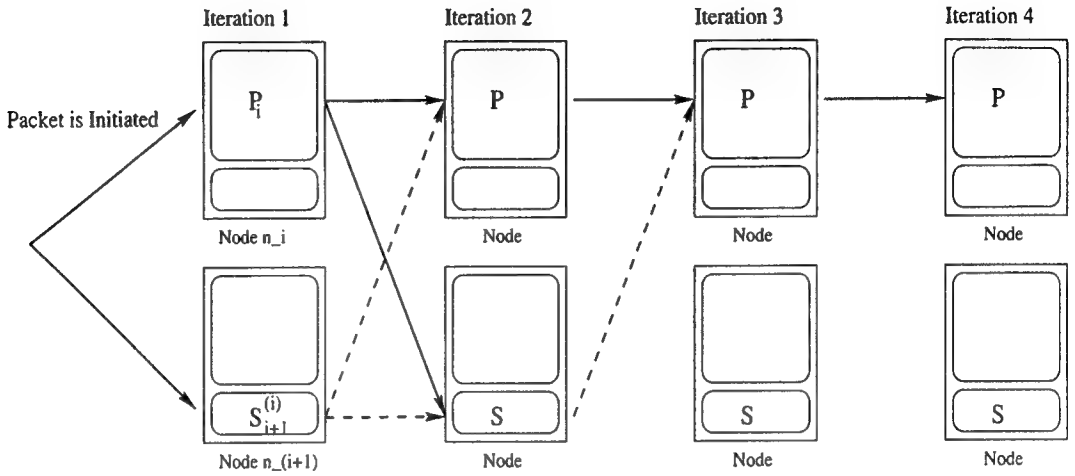


Figure 13: Path of packets as they travel around the fault tolerant system in either the radiosity or ray-tracing phase.

One such scheme follows. In both phases of the fault tolerant application, if a packet is below some specified iteration index, it is duplicated and sent to both the primary and secondary processors, otherwise it is just sent to the primary. In Figure 13, node n_i is primarily responsible for some part of the 3d object space, and node n_{i+1} is secondarily responsible for that part. In both phases, if the primary is active and receives a packet, computation is carried out as normal, and the packet is forwarded (if appropriate) to the subsequent primary-secondary pair. Packets are not sent to a secondary when beyond the first or second iteration. Further, packets will *never* be sent out by a secondary if its corresponding primary is operational. Beyond this, the operation of the secondary varies depending on whether we are in the radiosity phase or the ray-tracing phase.

In the Radiosity Phase, the secondary does in fact carry out computation on all of the packets it receives – the state information (regarding diffuse light values at each patch in its own sub-space) must be kept up to date with that of the primary. In the Ray-Tracing phase on the other hand, the secondary need only carry out computation when it knows that its corresponding primary is faulty and will not be able to update the light value at the pixel represented by the ray.

For recovery, if a primary is restarted, its state (the state of diffuse light in its sub-space) can be updated by the secondary serving as its backup.

Thus, each processor only need know the condition of the node for which it is backup. Global information regarding the condition of each node is not required. This more localized information could be obtained via "I'm alive" messages or perhaps through local or remote check routines to ensure (correct) operation.

We have thus far assumed that the first iteration is the one most likely to make a big visual impact on the quality of the scene image [18], and is therefore the most important. We show this to be the case with example images, rendered with varying ray-depths. In the test scene there are two tables, two windows, a mirror on the far wall, a chess board and pieces on the near table, and a glass sphere (to illustrate refraction) on the chess board. Both images were created and rendered using the Povray Ray-Tracing Software, however Figure 14 was rendered with ray-depth limited to one iteration, while Figure 15 was rendered with ray-depth limited to two iterations. Figure 16 was rendered with a depth of five iterations, which is the default maximum depth for Povray. As can be seen, the first two figures are of very similar quality as the most fully rendered image. Figure 14 lacks information

concerning the reflection in the mirror and the refraction visible within the glass sphere. While in Figure 15, a depth of two iterations, we see the reflection in the mirror and a bit more of the refraction within the sphere. Given this, we see that if a primary process is down, and the secondary is only allowed to provide results for rays within their first iteration, we see that we do not lose a great deal of image quality.

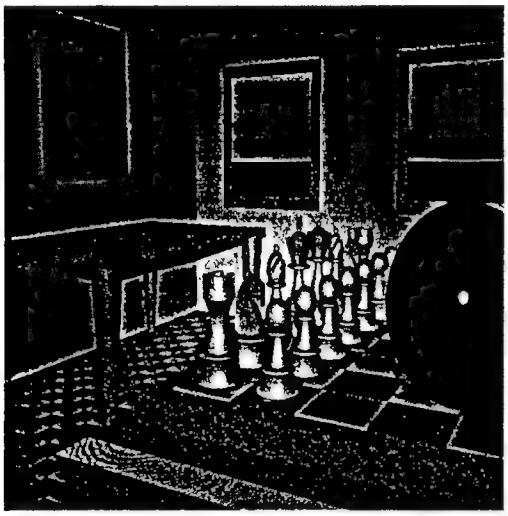


Figure 14: The test scene, rendered with a ray-depth of one.

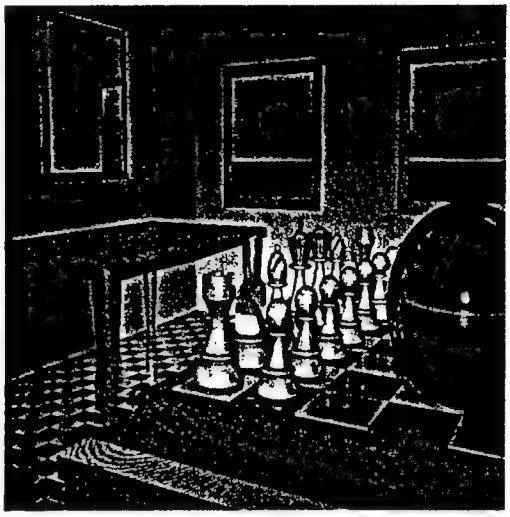


Figure 15: The test scene, rendered with a ray-depth of two.

We now consider the computational overhead it imposes. We have analyzed this in a simple simulation of the application. The simulation entailed "shooting" rays through a 64×64 array of pixels, and giving each ray a 35% chance of intersecting with an object in each sub-space it traveled through. Within that, there was a 35% chance of intersection with a object causing reflection and refraction (two rays result) and a 45% chance of reflection only (one ray results), and a 20% of termination (due to landing in a shadow or some other reason). Computation time for ray-packets under each of the above circumstances comes from [19].

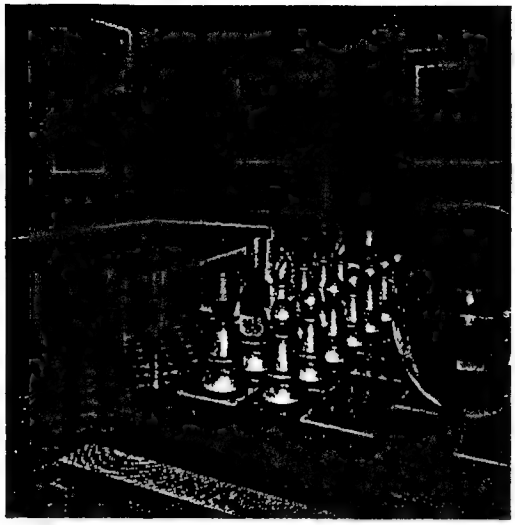


Figure 16: The test scene, rendered with a ray-depth of five.

Further, we assumed that the secondary not only receives the packet (incurring some overhead), but also always performs computation. Naturally, this will not generally be the case, as the primary will hopefully not (always) be down. The simulation conveys an idea of the worst case load, imposed when the corresponding primary is faulty.

Figure 17 demonstrates that our technique will impose only a light computational load on the system. If the secondary only carries out computation for the first 20% of the iterations (only 1 out of a possible 5 iterations for example), we see that the computational load imposed is less than 20%. The slight non-linearity of this relationship is due to the fact that there is a high probability of each subsequent iteration involving more and more packets as secondary rays are created due to reflection and refraction. This is encouraging for our technique because those first two iterations are likely to have a relatively large impact on the quality of the resulting image.

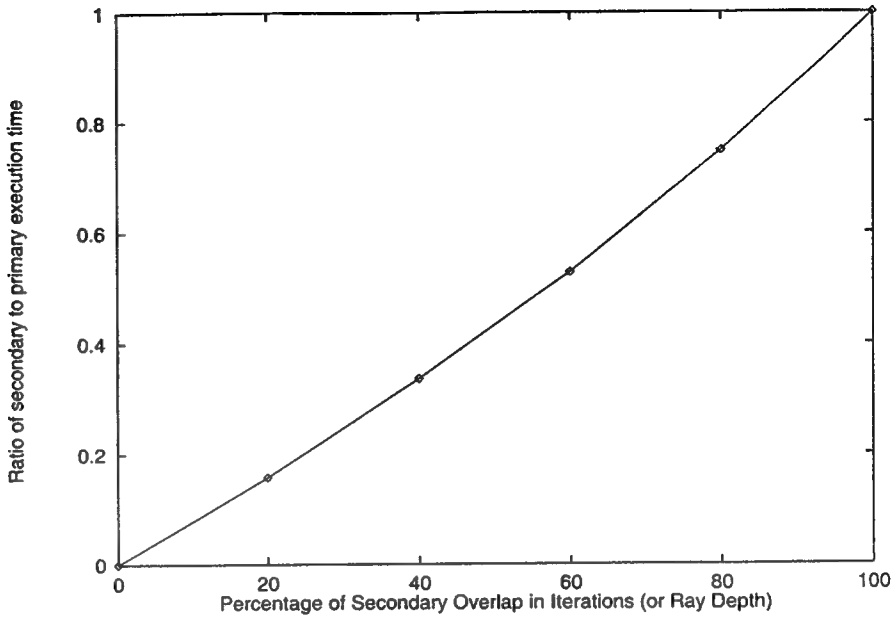


Figure 17: Ratio of secondary to primary execution time for our simulation of the 3d parallel rendering application.

We feel that a relatively similar result will hold for the radiosity phase as well, as a progressive refinement algorithm [20], similar to the ray-tracing computation, is employed there as well.

In analysis of the memory overhead imposed by our technique, we see that we are, unfortunately, assuming up to a 100% duplication, so that each processor can maintain information about both the sub-space that it is primarily responsible for, and that for which it is secondarily responsible. In terms of communication, we see that in those iterations where the secondary might be required to do some work a duplicate packet is sent to that secondary. The overhead here will vary depending on the number of iterations for which the secondary will overlap.

CHAPTER 6

FUTURE WORK

There are many ways in which we can further develop application-level fault tolerance that would be quite helpful to the engineering community. Two main areas are: 1) Further analysis of the costs of Application-Level Fault Tolerance, and 2) Extension of ALFT towards Algorithm-Based Fault Tolerance.

Toward the first goal one would want to conduct a thorough survey of the amount of memory and communicational overhead imposed by the secondary task set with a variety of applications. Here we have only considered the computational overhead. Memory overhead adds to the cost of ALFT in that each processing node might require some extra memory in order to be able to run both the secondary and primary tasks. Extra communication adds both to the load on the network and to the time required to complete one period or iteration of the Primary and Secondary task sections.

Regarding bridging ALFT and Algorithm-Based Fault Tolerance, ALFT is best in situations when faults can be dealt with on a process-by-process or node-by-node basis, but that Algorithm-Based Fault Tolerance is best when faults/errors should be dealt with on a bit-by-bit or primitive element basis. It might be useful to be able to add some of the capability of ABFT to ALFT (or vice-versa) such

that when a fault or error is detected, the application can decide to either attempt to correct the error using ABFT-like techniques, or to make use of ALFT-like techniques to use a secondary set of results. In order to do so, it would be necessary to provide fault/error detection schemes such that the distinction could be made between when an error can be corrected and when an error is uncorrectable and the secondary should be used.

CHAPTER 7

CONCLUSION

A high degree of fault tolerance may be obtained with a small investment of system resources in applications exhibiting data parallelism between two or more parallel processes. It is achieved through a combination of application-level and system-level fault tolerance. Application-Level Fault Tolerance consists of duplicating some portion of each parallel node's work on a neighboring node. These additional duties are known as the secondary task set.

The computational overhead of the secondary can be reduced by prioritizing data set elements and/or reducing the granularity of the dataset so that the secondary operates upon fewer elements. The secondary is not meant to completely replace the primary (original) process, but is a short-term substitute for times when the primary is down. The primary might return when the fault disappears or if the system restarts the process on another good node.

The type of inter-process data dependency will affect how the secondary task section is integrated with the original application. If there is no dependency, then the secondary would be executed only after the primary. However, if there exist phases, where output of one phase is required by other nodes, then a node's primary and secondary duties might be interleaved. A continuous dependency will require

that the secondary and primary be able to execute on demand. The secondary task section need not always be executed. If a fault is detected, the priority of the secondary could be raised, to ensure that it will complete without missing its deadline, and provide the necessary data for output.

We stress that Application-Level Fault Tolerance is not a single fixed set of steps, but a technique whose implementation will vary from application to application. The most effective ALFT implementation will be the one that is best tailored to the application in question.

Our technique is a substantial improvement over complete system duplication, in that it does not require 100% system redundancy, but merely adds a small amount of load to the existing system in achieving the same amount of fault tolerance. It differs from the recovery block approach in that the secondary does not have to be cold-started, but is ready for execution when a failure of the primary is detected. In addition, the level of reliability may be varied by varying the amount of redundancy. Regarding Algorithm-Based Fault Tolerance, our technique addresses the situation when faults occur that render a node or process completely useless (for example, a complete disconnection). While Algorithm-Based Fault Tolerance is equipped to handle the situation when errors in data are detected but are correctable.

In order to integrate such Application-Level Fault Tolerance into a particular application, the designer will need to first decide how best to reduce the computational overhead of the secondary. Then, the designer should choose mechanisms by which the secondary gets the input data it needs, is able to compute and output

results when necessary, and is able to communicate with the primary for synchronization purposes. Naturally, some sort of fault detection will have to be used. We believe that steps to integrate this technique into the application should be taken right from the early stages of design in order for this approach to be most effective.

BIBLIOGRAPHY

[1] Pradhan, D.K. Fault-Tolerant Computer System Design. Upper Saddle River, New Jersey: Prentice Hall PTR, 1996.

[2] Siewiorek, D. and Swarz, R.
Reliable Computer Systems Design and Evaluation. 2nd ed. Burlington, MA: Digital Press, 1992.

[3] Huang, K. and Abraham, J.A., "Algorithm-Based Fault Tolerance for Matrix Operations," *IEEE Transactions on Computers*, vol. C-33, pp. 518-528, Jun. 1984.

[4] Banerjee, P. and Abraham, J.A., "Bounds on Algorithm-Based Fault tolerance in Multiple Processor Systems, " *IEEE Transactions on Computers*, vol. C-35, No. 4, pp. 296-306, April 1986.

[5] Luk, F.T. and Park, H., "An Analysis of Algorithm-Based Fault Tolerance Techniques," *Journal of Parallel and Distributed Computing*, vol. 5, pp. 172-184, 1988.

[6] Rozenkrantz, D.J. and Ravi, S.S., "Improved Bounds for Algorithm-Based Fault Tolerance," *IEEE Transactions on Computers*

[7] Gu, D., Rozenkrantz, D.J., and Ravi, S.S., "Determining Performance Measures of Algorithm-Based Fault Tolerant Systems," *Journal of Parallel and Distributed Computing*, vol. 18, pp. 56-70, 1993.

[8] Tao, D.L., Hartmann, C.R.P., and Han, Y.S., "New Encoding/Decoding Methods for Designing Fault-Tolerant Matrix Operations," *IEEE Transactions on Parallel and Distributed Systems*, vol. 7, pp. 931-838, Sept. 1996.

[9] Yajnik, S. and Jha, N.K., "Graceful Degradation in Algorithm-Based Fault Tolerance Multiprocessor Systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 8, pp. 137-153, Feb. 1997.

[10] Randell, B., System Structure for Software Fault Tolerance. *IEEE Transactions on Software Engineering*, vol. SE-1, pp. 220-232, 1975.

[11] VanVoorst, B., Jha, R., Pires, L., Muhammad, M., "Implementation and Results of Hypothesis Testing from the C^3I Parallel Benchmark Suite," in *Proceedings of the 11th International Parallel Processing Symposium*, 1997.

[12] Castanon, D.A., and Jha, R., "Multi-Hypothesis Tracking (Draft)", DARPA Real-Time Benchmarks, Technical Information Report (A006), 1997.

[13] Hamza, R., Honeywell Technology Center, "Sonar Adaptive Beamformer (Draft)", DARPA Real-Time Benchmarks, Primary Technical Information Report, 1998.

[14] Allalouf, M., Chang, J., Durairaj, G., Lakamraju, V.R., Unsal, O.S., Koren, I., Krishna, C.M., "RAPIDS: A Simulator Testbed for Distributed Real-Time Systems," *Advanced Simulation and Technology Conference*, 1998, pp. 191-196.

[15] Liao, W., Ou, C., and Ranka, S., "Dynamic Alignment and Distribution of Irregularly Coupled Data Arrays for Scalable Parallelization of Particle-In-Cell Problems," in *Proceedings of the 11th International Parallel Processing Symposium*, 1996, pp. 57-61.

[16] Lubeck, M. and Farber, V., "Modeling the performance of hypercubes: A case study using the particle-in-cell application," *Parallel Computing* Vol. 9, 1988/89, pp. 37-52.

[17] Kobayashi, H., Yamauchi, H., Toh, Y., and Nakamura, T., "A Heirarchical Parallel Processing System for the Multipass-Rendering Method," in *Proceedings of the 11th International Parallel Processing Symposium*, 1996, pp. 62-67.

[18] Whitted, T., "An Improved Illumination Model for Shaded Display," in *Communications of the ACM*, Vol. 23, Number 6, June, 1980, pp. 343-349.

[19] Kobayashi, H., Nishimura, S., Kubota, H., Nakamura, T., and Y. Shigei, "Load Balancing Strategies for a Paralle Ray-Tracing System Based on Constant Subdivision," in *The Visual Computer*, 1988, pp. 197-209.

[20] Cohen, M.F., Chen, S.E., Wallace, J.R., and Greenberg D.P., "A Progressive Refinement Approach to Fast Radiosity Image Generation," in *Computer Graphics*, Vol. 22, Number 4, Aug. 1988, pp. 75-84.

Appendix 7

EVALUATING THE RELIABILITY OF DISTRIBUTED REAL-TIME SYSTEMS

A Thesis Presented

by

GOPINATH DURAIRAJ

Submitted to the Graduate School of the

University of Massachusetts Amherst in partial fulfillment

of the requirements for the degree of

MASTER OF SCIENCE IN ELECTRICAL AND COMPUTER ENGINEERING

February 1999

Department of Electrical and Computer Engineering.

ABSTRACT

EVALUATING THE RELIABILITY OF DISTRIBUTED REAL-TIME SYSTEMS

FEBRUARY 1999

GOPINATH DURAIRAJ, B.E., R.E.C. TIRUCHIRAPALLI, INDIA

M.S., UNIVERSITY OF MASSACHUSETTS AMHERST

Directed by: Professor C. M. Krishna

Computers are increasingly being used in life-critical applications and their need to be reliable also increases dramatically. A distributed system architecture is a very attractive proposition to meet these reliability and fault tolerance requirements. Such a system is very complicated and needs to be validated before building and deploying it.

A simulator test bed is built at UMass to model a variety of such systems quickly from a few basic building blocks. Since these systems are very complex, many independent simulation runs are needed to estimate their reliability to a reasonable level of confidence.

Importance sampling is a technique commonly used to speed up such rare event simulations. In this technique, the probabilistic dynamics of the system are altered so that the rare events (such as the system failure) occur much more frequently. The sample outputs then need to be adjusted to compensate for the bias introduced.

This thesis talks about building the simulator test bed and concentrates on incorporating and validating importance sampling to speed up the reliability estimations.

Two importance sampling heuristics called 'forcing' and 'failure biasing' are incorporated in the test bed. The implementation is validated by comparing the reliability estimates with that of the normal simulation. The effect of the failure bias on the dynamics of the scheme are also investigated to provide some guidance on choosing the failure bias.

The tool is applied to see how the reliability estimates of a system changes with the change in failure rate. Finally the tool is used to demonstrate that using an optimal failure recovery algorithm can significantly improve the reliability of a distributed real-time system.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Computers are wonderful machines. They are increasingly being used in ways never imagined possible before. From their traditional uses in number crunching and huge databases they have evolved rapidly and are now used to control everything from cars to factories and fly-by-wire aircraft.

As computers begin to be used in these life-critical applications, their need to be reliable increases dramatically. These computers are expected to perform their tasks in a time-bounded fashion. It is not enough if these machines deliver outputs that are logically correct they should also be timely. Catastrophic results can occur if these task deadlines are not met (Imagine an aircraft not putting out its wheels when it is fast approaching the runway). These computers are also increasingly expected to perform their best even in the presence of faults. They should be able to tolerate faults in hardware, software or anywhere!

A distributed system architecture is a very attractive proposition to meet these reliability and fault tolerance requirements. It has multiple, independent computing entities, which provide scalable computing power. Since they are independent, there is a high chance that the faults will be isolated and localized to a subset of nodes. When a fault is detected in a node, the tasks from that node can be moved to other

active nodes that can handle the additional load. Such an architecture can also exploit the potential parallelism that might exist among the various units. Does it look like we have solved all the tough problems? Alas, there is no such thing as a free lunch! The distributed system architecture is very complex and there are quite a few issues to be solved.

- We need good task admission, allocation/scheduling algorithms to make sure that the accepted tasks meet their deadlines.

- We need a good resource management algorithm to efficiently manage the available redundancy. When a node fails, there are a few possible recovery actions. An optimal failure recovery action must be chosen so that the possibility of a task missing its deadline is minimized.

- We need a good network architecture to interconnect the distributed nodes and to provide high connectivity and performance even in the presence of faults. This network should also provide for predictable delays as the tasks might be exchanging messages that are time constrained. As the number of computing nodes are increased, the network should also be able to scale well.

Each of the above problems constitute a major research area in itself and a lot of work has gone into solving them. At UMass an optimal resource management algorithm [30] has been developed to suggest the failure recovery action that will minimize the probability of a critical task missing its deadline.

Once we have some solutions for each of the problems, we would like to put together a complete system and find out the reliability of the system to validate the

design. Such a system is very complex and the algorithms/policies might interact in ways that may not be obvious. The reliability evaluation is usually carried out by analytical techniques or simulation. Analytical modeling is very tough in all but the simplest of cases and simulation is the preferred way to evaluate complex models.

1.1 Thesis goal

We have endeavored to create a simulator test bed that will facilitate building such a complex system and to evaluate its reliability. The simulator test bed will enable the user to model a variety of systems quickly using a few basic building blocks. Adding new building blocks as plug-ins should also be easy. Examples of such building blocks include scheduling algorithms, allocation algorithms, failure recovery policies, new network types etc.

Once we have created the system under evaluation, we need efficient techniques to evaluate its reliability. Means should be found to integrate such techniques with the simulator test bed. This will enable the user to configure a system of choice and find out its reliability quickly.

Since the modeled systems are very complex and their reliability is quite high, many independent simulation runs will be needed to estimate their reliability to a reasonable level of confidence. Importance sampling is a technique commonly used to speed up such rare event simulations. In this technique, the probabilistic dynamics of the system are altered so that the rare events of interest occur much more frequently. The sample outputs then need to be adjusted to compensate for the bias introduced.

This thesis talks about building such a simulator test bed and concentrates on incorporating importance sampling in the test bed to speed up the reliability estimations.

1.2 Thesis Outline

The rest of the thesis is organized as follows: Chapter 2 talks about efficient simulation techniques for reducing the sample variance. Chapter 3 talks about the simulation model and some of the interesting issues involved in the design and implementation of such a simulator.

Chapter 4 clarifies some issues regarding reliability evaluation and gives an overview of the simulation analysis that will be used. Chapter 5 talks in detail about the implementation of importance sampling, its validation and the effect of the bias parameter.

CHAPTER 2

IMPORTANCE SAMPLING

Different measures are used to evaluate the modeled systems depending upon whether they are mission oriented systems or continuously operating systems. Some of the dependability measures that are very commonly used are steady-state availability, reliability, mean time to failure (MTTF), expected interval availability etc.

We will discuss a technique called Importance Sampling which is widely employed to speed up rare-event simulations in queueing and reliability models. A variety of heuristics have been proposed to implement importance sampling to estimate the different measures mentioned above. Depending upon the measure that we are interested in estimating, we could choose to implement some of them. Since we are primarily interested in measuring the reliability of the system, we will look closely at those heuristics that are applicable.

2.1 Introduction

To analyze the given system for reliability, we need to model the system in a way that closely mirrors reality and which makes the analysis simple. The system is considered to be a collection of components which can fail and possibly get repaired.

It is considered operational if at any given moment the operational components satisfy some minimum system operational requirements. The failure times and the repair times of the components are assumed to be exponentially distributed so that the system may be modeled as a continuous time Markov chain (CTMC).

Typically numerical methods are used to solve Markov chains. Although many modeling packages have been built (eg [10]), the size of the system modeled is typically small because the number of states in the system increases exponentially with the number of components. Techniques such as state lumping and unlumping and state aggregation and bounding can reduce the size of the state space substantially. However large systems with a large number of redundant components are still out of the range of the solution capabilities of current numerical methods primarily due to storage or computational limitations.

An alternative approach for the solution of large models is Monte Carlo simulation. By nature this approach has the immediate advantage of having relatively small storage requirements. In addition it might be easier to model very complex systems using simulation. On the other hand, since the failure events are very rare, it is apparent that the analysis by simulation of large models with a high degree of redundancy will require many long independent replications in order to attain reasonable confidence intervals. Our goal is to investigate the variance reduction methods that might be easily implemented in the current model.

In importance sampling, we change the probabilistic dynamics of the system for simulation purposes. The new probability measure introduces system failures to occur more frequently. We then make adjustments to the sample outputs to unbias the

estimator before computing it. A good reference for the mechanics of this technique is Hammersley and Handscomb [13]. Generalizations to stochastic systems are given in Glynn and Iglehart [9]. The heuristic of failure biasing was first proposed in Lewis and Bohm [18] in the context of reliability estimation of nuclear reactors. In Goyal, Heidelberger and Shahabuddin [11] it was adapted to the estimation of unavailability for highly reliable Markovian systems. In Shahabuddin et al. [23], it was used for the estimation of the MTTF using a regenerative method. Generalizations of these heuristics along with new ones have been investigated in the unifying paper Goyal et al [12].

2.2 Basic Idea

Let X be the random variable that has the probability density function $p(x)$. We are interested in estimating the probability θ that X is in some set A (the failed state)

$$\theta = \int_{-\infty}^{+\infty} 1_{\{x \in A\}} p(x) dx = E_p[1_{\{X \in A\}}]$$

where the subscript p denotes sampling from the density p and $1_{x \in A}$ is the indicator of the set A. i.e.,

$$1_{\{x \in A\}} = \begin{cases} 1 & \text{if } x \in A \text{ and} \\ 0 & \text{Otherwise} \end{cases}$$

Consider estimating θ by simulation. The standard approach would be to draw n samples $X_1, \ldots, X_n$ from the density p, set $I_i = 1_{X_i \in A}$ and form the estimate

$$\overline{X} = \frac{1}{n} \sum_{i=1}^{n} I_i$$

If $E(X^2) < \infty$ then using the central limit theorem, we can construct the standard confidence interval on the estimator. This confidence interval is given by $[\overline{X} - HW, \overline{X} + HW]$ where HW is called the half width and is given by $z_{\alpha/2}S/\sqrt{n}$ (the quantity $z_{\alpha/2}$ is the $100(1-\alpha/2)$ percentile point of the standard normal distribution and S is the standard deviation of the estimator). **The relative error (RE)** is defined to be HW/θ.

Multiplying and dividing the integrand by another density function $p'(x)$, we obtain

$$\begin{aligned} \theta &= \int 1_{\{x \in A\}} \frac{p(x)}{p'(x)} p'(x) dx \\ &= E_{p'}\left[1_{\{X \in A\}} \frac{p(X)}{p'(X)}\right] \\ &= E_{p'}[1_{\{X \in A\}} L(X)] \end{aligned} \tag{2.1}$$

where $L(x) = p(x)/p'(x)$ is called the **likelihood ratio** and the subscript p' denotes sampling from the density p'. The above equation is valid for any density p' provided that $p'(x) > 0$ for all $x \in A$ such that $p(x) > 0$. i.e., a non-zero feasible sample under density p must also be non-zero feasible sample under p'.

The above equation is the key for importance sampling. Draw n samples $X_1, \ldots, X_n$ using the density P' and define $\delta_i = L(X_i)I_i$. Then by equation 2.1 $E_{p'}[\delta_n] = \theta$. Thus an unbiased estimate of θ is given by

$$\overline{X}(p') = \frac{1}{n}\sum_{i=1}^{n} \delta_i = \frac{1}{n}\sum_{i=1}^{n} I_i L(X_i)$$

i.e., θ can be estimated by simulating a random variable with a different density and then unbiasing the output (I_i) by multiplying by the likelihood ratio. Sampling with a different density is sometimes called a "change of measure" and the density p' is called the importance sampling density.

2.3 Choosing an Optimal Sampling Density

Since essentially any density p' can be used for sampling, what is the optimal density i.e., what is the density that minimizes the variance of $\overline{X}(p')$? Selecting $p'(x) \equiv p^*(x)$ as follows

$$p^*(x) = \begin{cases} p(x)/\theta & \text{for } x \in A \text{ and} \\ 0 & \text{Otherwise} \end{cases}$$

has the property of making $\delta_i = I_i P(X_i)/P^*(X_i) = \theta$ with probability one. Since the variance of a constant is zero, $p^*(x)$ is the optimal change of measure. However, there are several practical problems with trying to sample from this optimal density p^*. First, it explicitly depends upon θ, the unknown quantity that we are trying to estimate. If in fact θ were known, there would be no need to run the simulation experiment at all. Second, even if θ were known, it might be impractical to sample efficiently from p^*.

Since the optimal change of measure is not feasible, how should one go about choosing a good importance sampling change of measure? Since $E_{p'}[\delta_i] = \theta$ for any density p', reducing the variance of the estimator corresponds to selecting a density

p' that reduces the second moment of δ_i

$$\begin{aligned} E_{p'}[Z_i^2] &= E_{p'}\left[(I_i L(X_i))^2\right] \\ &= \int 1_{x\in A}\left(\frac{p(x)}{p'(x)}\right)^2 p'(x)dx \\ &= \int 1_{x\in A}\frac{p(x)}{p'(x)}p(x)dx \\ &= E_p[I_i L(X_i)] \end{aligned}$$

Thus to reduce the variance, we want to make the likelihood ratio $p(x)/p'(x)$ small on the set A. Since A is a rare event, roughly speaking, $p(x)$ is small on A. Thus to make the likelihood ratio small on A, we should pick p' so that $p'(x)$ is large on A. i.e., the change of measure should be chosen to make the event A more likely to occur.

Importance sampling does not always lead to a reduction in variance and can produce arbitrarily bad results if not applied carefully. Essentially all work on using importance sampling in practical applications deals with choosing an importance sampling distribution that leads to actual variance reduction.

2.4 Likelihood Ratio

In order to apply importance sampling, it must be possible to compute the relevant likelihood ratio. Essentially, each time a change of measure is performed, the likelihood ratio for that variable is computed (using the original and new densities) and incorporated into a running product.

In the case of sampling from a single probability density function as described above, the likelihood ratio $L(X) = P(X)/P'(X)$ This equation is also valid if X is drawn from a discrete distribution. i.e., if

$$\begin{aligned} P(X = a_i) &= P(a_i)\ i = 1, \ldots, n \text{ and} \\ P'(X = a_i) &= P'(a_i)\ i = 1, \ldots, n \end{aligned}$$

We require that $P'(a_i) > 0$ if $P(a_i) > 0$ but note that we can have $P'(a_i) > 0$ even if $P(a_i) = 0$ since the likelihood ratio is zero in this case. i.e., no weight is given to an impossible (under p) sample path. Suppose $\mathbf{X} = (\mathbf{X_1}, \ldots, \mathbf{X_n})$ is a random vector where X_i is drawn from density $P_i(x)$ and X_i is independent of $X_j (j \neq i)$. If under importance sampling, X_i is drawn from density $P_i'(x)$ and again X_i is independent of $X_j (j \neq i)$, then

$$L(X) = \prod_{i=1}^{n} \frac{P(X_i)}{P'(X_i)}$$

Suppose $\{X_i, i \geq 0\}$ is a discrete time Markov chain (DTMC) on the state space of non-negative integers where X_0 has the (initial) distribution $P_0(i)$ and the step transition probabilities are given by $P(i,j) = P(X_m = j | X_{m-1} = i)$. Let $\mathbf{X_m} = (\mathbf{X_0}, \ldots, \mathbf{X_m})$. If, under importance sampling, X_0 is drawn from $P_0'(i)$ and the process is generated with the one-step transition probabilities $P'(i,j)$ then

$$L(X_m) = \frac{p_0(X_0)}{p_0'(X_0)} \prod_{i=1}^{m} \frac{P(X_{i-1}, X_i)}{P'(X_{i-1}, X_i)}$$

2.5 Failure Biasing

In the last few sections, we have seen the theoretical foundations of importance sampling. There are quite a few heuristics for the implementation of this technique, applicable for estimating different dependability measures. Failure biasing is an important heuristic used heavily in the estimation of transient measures such as reliability, mean time to failure etc.

For estimating the transient measures, we would like to apply importance sampling so as to sample most often from the most likely paths to failure. However, in complex systems, it may not be easy to identify these most likely paths. Thus we need to develop heuristics that are simple to implement and search many different paths to failure. It should sample often enough from the most likely failure paths so as to obtain variance reduction. Failure biasing is designed to do this.

The objective of the biasing is to derive results with reduced variance. This can be accomplished by causing more trials to contribute to the result than in the binomial case. As this happens, there will be fewer zero weight trials, but those that do contribute will have weights much smaller than one.

Failure biasing has a few variants that work well for certain types of systems. We will now look at a few important ones.

2.5.1 Simple Failure Biasing

Simple failure biasing was introduced in Lewis and Bohm [18]. At each transition of the system state, it biases the system towards more faults so as to drive the system

to the failure state. Whenever such a bias is made, the corresponding likelihood ratio is calculated and updated into a running product.

Let the failure and repair rates of a component i be exponentially distributed with rates λ_i and μ_i respectively. The total failure rate out of a particular state is the sum of the failure rates of the currently active components. Let us assume this is λ. Similarly the total repair rate out of the state is the sum of the repair rates of the components that are suffering a temporary fault. Let us call this quantity μ. γ is the total transition rate out of the current state and is equal to the sum of λ and μ.

In this technique, the probability of an additional failure is set to be a fixed probability ϕ. Typically ϕ is chosen to be in the range $0.20 \leq \phi \leq 0.80$. Note that in the original system the probability of this event is λ/γ. Since the repair rates are usually orders of magnitude greater than the failure rates, this ratio is very small. Thus the probabilistic dynamics of the system are altered with a heavy bias towards more failures.

When a failure transition is selected, the component getting the failure j is selected proportional to the original transition rates, i.e., with probability λ_j/λ. If a repair transition is selected, with probability $(1 - \phi)$, then the component getting repaired j is selected with probability μ_j/μ.

2.5.2 Balanced Failure Biasing

The above mentioned technique of simple failure biasing works very well for balanced systems. While the simple failure biasing takes the system to the set of failure

states with reasonable probability, it does not push the system along the more common failure paths often enough.

Consider a system with two types of components. There is one component of type 1 that has failure rate ϵ^2 and three components of type 2 that each have failure rate ϵ. The system is considered operational if at least one component of each type is operational. Under simple failure biasing, it is a type 1 failure with probability $\epsilon^2/(N_2\epsilon+\epsilon^2)$ where N_2 is the number of operational type 2 components; this probability is of order ϵ. Similarly, the probability of a type 2 failure is $(1 - O(\epsilon))$.

Thus, under simple failure biasing, when the system ends up in a failure state, most of the time it gets there by having three type 2 failures. It only rarely ends up in a state in which component 1 is failed. However, the path with a single component 1 failure is the most likely path to system failure; its probability is of order ϵ, whereas any other system failure path has a much smaller probability of the order ϵ^2. Thus, while simple failure biasing takes the system to the set of failure states with reasonable probability, it does not push the system along the right failure path often enough. The result of this is that simple failure biasing applied to unbalanced systems may result in estimates having unbounded relative error.

To overcome this, another variant of failure biasing called "balanced failure biasing" was introduced [24] and was shown to result in bounded relative error.

In balanced failure biasing, a failure transition is again chosen with a probability ϕ. Now however, given that a failure event has occurred, the probabilities of all failure transitions are equalized, i.e., a failure transition from i to j is conditionally selected with probability $1/F(i)$ where $F(i)$ is the number of failure transitions possible out

of state i. The selection of the repair transition is similar to the case of simple failure biasing.

Under balanced failure biasing, many unlikely paths to system failure may be generated, but enough of the most likely such paths are generated so as to guarantee good estimates. Further analysis that characterizes when these and more general failure biasing schemes are efficient is given in Nakayama [19].

2.6 Forcing

At each transition step of the simulation, we essentially need to make two decisions.

- When to have the next transition?
- What should be the next state?

The technique of failure biasing influences the second decision, i.e., we force the system towards additional faults. The technique of "forcing" [18] influences the first decision. It increases the probability of the system having another transition before the end of the mission time.

This technique is widely used to estimate transient measures such as unreliability, mean time to failure etc. Note that this heuristic is essentially independent of failure biasing and each of them can be applied in isolation. The likelihood ratios due to of them are calculated separately and multiplied together to give the cumulative likelihood ratio.

This technique is applied only when the mission time is sufficiently small that the probability of another transition occurring before the end of the mission time is

very small. When failed components are present, the total transition rate is usually sufficiently large that this heuristic need not be applied.

2.7 Other Importance Sampling Results

The above mentioned techniques of failure biasing and forcing are the most widely used importance sampling heuristics. These are also important to us as they are heavily used in estimating the reliability of systems. There are a few other results related to importance sampling and we will look at some of them here.

Regenerative Simulation

Some times, we are interested in the steady state performance measures such as the steady-state unavailability, i.e., the long run fraction of the time that the system is in a failure state. If the system is regenerative then the regenerative method can be used to estimate steady state performance measures.

Let X_s be the process at time s. We assume there is a particular state, call it 0, such that the process returns to state 0 infinitely often and that, upon hitting state 0, the stochastic evolution of the system is independent of the past and has the same distribution as if the process were started in state 0.

Let β_i denote the time of the i-th regeneration ($\beta_0 = 0$) and $X_0 = 0$. Let $\alpha_i = \beta_i - \beta_{i-1}$ denote the length of the i-th regenerative cycle. If $E[\alpha_i] < \infty$ then under certain regularity conditions, $X_s \Rightarrow X$ as $s \to \infty$ (where $\Rightarrow$ denotes convergence in distribution and X has the steady state distribution). Let h be a function on the

state space and define $Y_i = \int_{\beta_{i-1}}^{\beta_i} h(X_s)ds$ Then $(Y_i, \alpha_i), i \geq 1$ are i.i.d. and

$$E[h(X)] = \frac{E[Y_i]}{E[\alpha_i]}$$

The above equation forms the basis of the regenerative method; simulate N cycles and estimate $E[h(X)]$ by $\overline{Y}_N/\overline{\alpha}_N$ where $\overline{Y}_N$ and $\overline{\alpha}_N$ are the averages of the Y_i-s and α_i-s respectively.

For Large Time Horizons

When the mission time is fixed, balanced failure biasing and forcing produce bounded relative error estimates of the unreliability $U(t)$. For very large time horizons, the empirical effectiveness of forcing decreases and a somewhat different approach has to be taken.

This problem is analyzed in Shahabuddin [26] and the regenerative structure of the system is exploited to estimate tight upper and lower bounds on $U(t)$. A different approach to estimating $U(t)$ is presented in Carrasco [5]. Instead of estimating $U(t)$, its Laplace transform $\hat{U}(s)$ is estimated at a number of values of s. The regenerative structure is again exploited by deriving a renewal equation for $\hat{U}(s)$ in terms of quantities defined over a single cycle that can easily be estimated. Numerical inversion of the estimated Laplace transform is then used to recover estimates of $U(t)$.

Failure Distance Biasing

Carrasco [4] considers another failure biasing approach, termed "failure distance biasing" which attempts to improve on the efficiency of balanced failure biasing by

giving more weight to sample paths that are closer to the set of system failure states F.

In this approach, failure transitions are grouped into classes based on their estimated distance to F and more weight is given to the classes corresponding to shorter distances. Once a class is chosen, the individual transitions can be chosen either proportional to their original rates (unbounded relative error may occur in unbalanced systems) or they can be equalized (as in balanced failure biasing, then the resulting estimate has bounded relative error).

In practice, the success of this approach depends on the ability to correctly and efficiently assign the distances. The class of systems for which this can be done is unclear. In some cases, significant improvements over balanced failure biasing have been obtained for systems with a large number of component types.

2.8 A Sample Application

We will now look at a practical application of the importance sampling for evaluating the reliability of a complex system. Boyd and Bavuso [3] talk about the modeling of a highly reliable fault-tolerant guidance, navigation and control system for long duration spacecraft. It is of considerable interest to us, as it is the type of system that we aim to model using the simulator.

System Model

The system consists of a 3-dimensional hypercube configured as two fault-tolerant 2-dimensional modules, each with a spare processing node. This spare could be a

hot or cold spare. Each processing node communicates with the other processing nodes in the system through four ports. For the system to be operational all eight processing nodes must be operational and must all be able to communicate with each other. Therefore, the system will be considered failed if any processing node fails and a spare node is unable to take over or if any two nodes in the hypercube are unable to communicate with each other.

The mission time of the system was assumed to be 10 years. It was clear from preliminary studies that the system with a traditional constant failure rate model will not meet the high reliability requirements. More recently acquired empirical data provided evidence that decreasing fault rates are common in spacecraft applications. Hence, they were interested in the effect of assuming that the components having a Weibull decreasing fault rate instead of the usual constant failure rate that is characteristic of the time-homogeneous Markov models. They were also interested in assessing the improvement in system reliability, if any, that can be achieved by using a cold spare processor in the processing nodes instead of a hot spare.

Simulation

The inclusion of decreasing failure rates with cold spares requires the use of a non-Markovian reliability model which is substantially more difficult to solve than the Markovian model that assumes a constant failure rate. Since the analytical solution of the system was extremely tough it was planned to use simulation.

Since the system failure events are extremely rare, a large number of trials will be needed to evaluate the reliability. To speed up the reliability evaluation, impor-

tance sampling techniques such as forced transitions and simple failure biasing were implemented.

Results

Results using constant failure rates indicated that the proposed architecture would be inadequate, with the probability of system failure exceeding 60%. Initial attempts to evaluate the model with HARP [6] (which uses analytical solution techniques) were not successful due to the large size of the model.

The simulator was able to handle the system model well. The effect of assuming Weibull decreasing fault rate clearly resulted in decreasing system unreliability. There was a difference of about three orders of magnitude in the system unreliability from 0.631 ± 0.013 when all the components have constant fault rate to about $0.777 \times 10^{-3} \pm 0.41 \times 10^{-3}$ when all the components have Weibull distribution.

From their experience in implementing the importance sampling for a complex model they have the following recommendations: Simulation techniques such as importance sampling are able to evaluate models that are beyond the reach of analytical techniques both in terms of memory and execution time. If only ballpark estimates are desired, simulation may be able to produce the required results relatively quick. However, if the accuracy of the evaluation is important, the execution time required by the simulation increases rapidly. Therefore, they advocate using the analytical methods if this is feasible and use simulation to evaluate the more complex cases.

2.9 Other Variance Reduction Techniques

Importance sampling is not the only variance reduction method used in simulation. There are quite a few other techniques that could be used for variance reduction. We will here try to present a sampling of such schemes. For an excellent discussion of the topic see [7] and [22].

Common Random Numbers

Common random numbers method is normally used when estimating the difference between the expected performance measures of two or more systems. It is perhaps the most widely used variance reduction method in practice. Suppose we want to estimate $\eta_1 - \eta_2$, where η_1 and η_2 are two unknown quantities, estimated by X_1 and X_2 respectively. Let $Z = X_1 - X_2$ and suppose that $E[Z] = \eta_1 - \eta_2$. The variance of Z is then

$$Var[Z] = Var[X_1] + Var[X_2] - 2Cov[X_1, X_2]$$

If X_1 and X_2 are generated independently, the covariance term disappears. But if we manage to introduce a positive covariance between X_1 and X_2 without changing their individual distributions, then the variance of Z will be reduced. The standard way of inducing such a covariance is to use the same underlying random numbers to drive the simulation for both X_1 and X_2 and to make sure that these random numbers are used at exactly the same place for both systems

Antithetic Variates

The idea of antithetic variates resembles that of the common random numbers mentioned above. Now, we want to estimate a single mathematical expectation η, using a pair of unbiased estimators (X_1, X_2). The unbiased estimator of η will be the average $X = (X_1 + X_2)/2$. Whose variance is given by

$$Var[X] = \frac{Var[X_1] + Var[X_2]}{4} + \frac{Cov[X_1, X_2]}{2}$$

If the $Cov[X_1, X_2] < 0$, then X has a smaller variance. A standard way of inducing the negative correlation is to use a sequence of underlying iid uniforms $\omega \equiv \{U_k, k \geq 1\}$ to drive the simulation for computing X_1 and use the antithetic sequence $1 - \omega \equiv \{1 - U_k, k \geq 1\}$ to drive the simulation when computing X_2. The rationale is that disastrous events in the first simulation should be compensated by the "antithetic" lucky events in the second one, thus reducing the variance of the average.

Conditional Monte Carlo

The general idea of Conditional Monte Carlo (CMC) is to replace the estimator $X = h(\omega)$ by its conditional expectation given another random variable Z. Roughly, if Z contains much less information than X, then the CMC estimator $X_{cm} \equiv E[X|Z]$ should have much less variability than X. More specifically, one has $E[X_{cm}] = E[X]$ and $Var[X_{cm}] = Var[X] - E[Var[X|Z]]$, so the variance can only decrease.

Indirect Estimation

Suppose that the mean μ of interest can be expressed as a known function of some other quantity η, say $\mu = f(\eta)$. Then it may be more efficient to estimate η instead of μ, then apply f to the estimator of η. This is called indirect estimation.

Assuming that we want to estimate the steady-state average queue length L_q. For the standard estimator, we simulate the system for a long time horizon and take the sample time-average. An alternative indirect estimator is based on Little's law $L_q = \lambda w_q$ where λ is the arrival rate and w_q is the average waiting time in the queue.

CHAPTER 3

IMPLEMENTATION OF THE TEST BED

As mentioned earlier, we want to build a simulator test bed where the user can configure a system of his/her choice and find out its reliability. This simulator should be modular so that it will be easy in future to modify or add new building blocks of the system. In this chapter we will take a close look at the simulation model, the overall design of the simulator and some interesting issues in building such a distributed test bed.

3.1 Simulation Model

Before we set out to design and build the simulator, we need to look at the system model that we are trying to simulate. A distributed real-time system can often be structured as a collection of computing nodes that are connected by a communication network. This can be modeled as illustrated in Figure 1.

From this model, we can identify some of the key elements of the system. They are

- The computing nodes

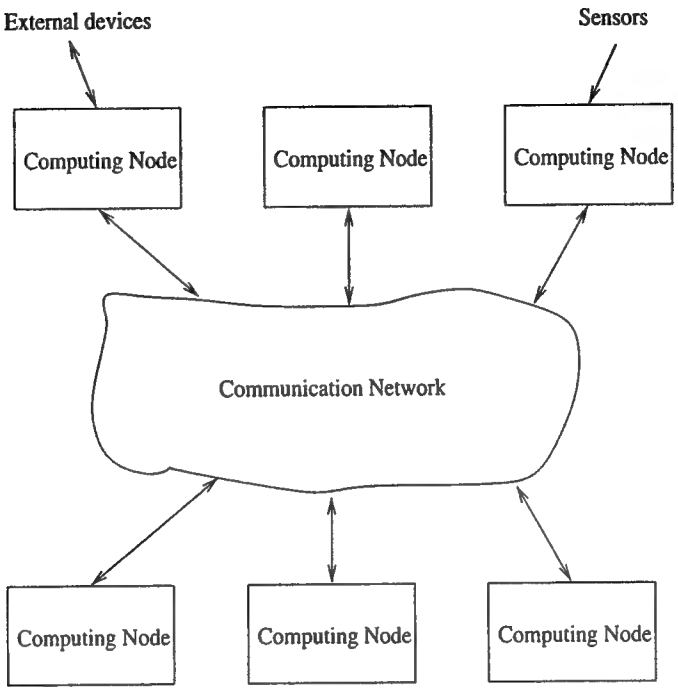


Figure 1: A typical distributed real-time system

- The tasks that run on the nodes
- The algorithms to be used for allocation/scheduling
- The messages that are exchanged by the nodes as part of execution or maintenance
- The network interconnect

We will take a look at each of these entities and see how they can be effectively modeled.

3.1.1 Computing Nodes

Each of the computing nodes in the system can be modeled as a single entity that is self sufficient in itself. It has private memory, it's own scheduling algorithm, checkpointing scheme, etc. Each of these can be individually changed without affecting the others. Once we have such a collection of nodes, we need to have some mechanism by which they work together. One of the most common solutions is to have a master-slave relationship among the nodes.

There is a single master node in the system whose only responsibility is to make sure that the rest of the nodes work together to execute the tasks even in the presence of faults. The responsibilities of the master can be enumerated as follows

- Task allocation. The jobs arrive from the outside world at the master which has to send it out to the appropriate node for execution.
- Fault Recovery. When there is a fault on one of the nodes, the recovery policy has to be consulted to choose the recovery action. The master then has to implement this action to the best of its ability.

To make this scheme work, the slave nodes have to periodically record their state in a checkpoint and send out 'alive' messages that the master can monitor. They also have to monitor the working of the master. In case the master malfunctions, a new master node has to be elected from the currently active slaves. Of course, this means that any slave node will have to be able to take on the master functionality smoothly.

3.1.2 Tasks and Messages

Many real-time applications and systems are highly structured, much more so than the general purpose systems. The set of tasks and their properties are generally known beforehand at least approximately. These tasks can be classified by nature of their recurrence. They are

- *Periodic Tasks:* Many tasks in real-time systems are executed periodically. For example, most of the monitoring tasks are periodic and these are executed at regular intervals. The periodicity of these tasks are known to the designer and so can be prescheduled.

- *Aperiodic Tasks:* These are tasks that occur only occasionally. The arrival pattern cannot be predicted.

- *Sporadic Tasks:* Aperiodic tasks with a bounded inter arrival time are called as sporadic tasks.

In a typical real-time system, there will be a mixture of these task types, the majority of them being periodic. In the current version of the simulator, we have allowed periodic tasks which can be described by attributes such as period, deadline, phase, redundancy, priority, messages to be sent and received during its execution etc. User could also insert a task during the simulation run.

The computing nodes exchange messages among themselves either as a result of the tasks that they are executing or as a part of the maintenance functions. Depending upon the functionality, they can be classified as data messages (which are sent as part

of the processing for the tasks) and control messages (which are used to monitor the state of the system etc).

3.1.3 Scheduling and Allocation Algorithms

Given a list of tasks that are defined as mentioned above, it is the task of the allocation/scheduling algorithms to make sure that the task deadlines are met. These algorithms can be characterized by the following parameters.

- Hard real-time (needs tough performance guarantees) versus soft real-time (can live with a best efforts approach)
- Preemptive (allows tasks to be suspended temporarily when a higher priority task arrives) versus non-preemptive scheduling (runs each task to completion)
- Dynamic (scheduling decisions are made during execution) versus static (scheduling decisions are made in advance)
- Centralized (one node collecting information and making the decisions) versus decentralized

Given a set of tasks, task precedence constraints, resource requirements, task characteristics and deadlines, the real-time computer has to come up with a feasible allocation/schedule. A task assignment/schedule is said to be feasible if all the tasks start after their release times and complete before their deadlines.

Allocation Algorithms

On systems with more than two processors, the task assignment/scheduling problem is NP-complete [16]. So, we use the following heuristic : First the master assigns the tasks to the processors and then runs the uniprocessor scheduling algorithm for each of the slave nodes to see if the allocation was feasible. If one or more of the schedules are infeasible, we must either return to the allocation step and change the allocation or declare that a schedule cannot be found.

Different allocation algorithms are used. They vary from a pure round-robin allocation to schemes which aim to keep the utilization of the individual processors below a limit. This limit depends on the characteristics of the tasks and the uniprocessor scheduling algorithm being run on the individual processors.

Uniprocessor Scheduling Algorithms

These algorithms typically assign priorities to the tasks and execute the task with the highest priority. The relative priorities of the tasks are a function of the nature of the tasks themselves and the current state of the system. The following two algorithms are implemented in the simulator.

Rate-Monotonic is an optimal and popular static-priority algorithm. It is used to schedule periodic, preemptible tasks whose deadlines equal the task period. The basic idea of the rate monotonic algorithm is to assign different and fixed priorities to tasks with different execution rates, highest priority being assigned to the highest frequency tasks. At any time, the low-level scheduler simply chooses to execute the

highest priority task. A task set of n tasks is schedulable under RM if its total processor utilization is no greater than $n(2^{1/n} - 1)$.

EDF is an optimal dynamic-priority scheduling algorithm. It is used to schedule preemptible tasks. The task with the earliest deadline has the highest priority. If a task set is not schedulable on a single processor by EDF, there is no other uniprocessor scheduling algorithm that can successfully schedule that task. If all the tasks are periodic and have relative deadlines equal to their periods, the test for task-set schedulability is simple: If the total utilization of the task set is no greater than 1, the task set can be feasibly scheduled on a single processor by the EDF algorithm.

3.1.4 Network

Communication in real-time distributed systems is different from communication in other distributed systems. While high performance is always welcome, predictability and determinism are the real keys to success.

Achieving predictability in a distributed system means that communications between processors must also be predictable. LAN protocols that are inherently stochastic, such as Ethernet are unacceptable because they do not provide a known upper bound on transmission time. As a contrast, consider a token ring LAN. Whenever a processor has a packet to send, it waits for the circulating token to pass by, then it captures the token and sends its packet. Assuming that each of the k machines on the ring is allowed to send at most one n byte packet per token capture, it can be guaranteed that an urgent packet arriving anywhere in the system can always be

transmitted within kn byte time plus overhead. This is the kind of upper bound that a real-time distributed system needs.

A faithful simulation of the interconnect network is crucial in getting a realistic cost of the message passing, task migration etc. It affects the estimation of overheads involved in doing the recovery and reconfiguration actions and consequently, the performance of the complete system.

The total delay D experienced by a packet (i.e., the time interval between the packet arrival at the network and its delivery to the destination) is given by $D = W + S + T$ where

W is the time spent waiting in the queue before starting transmission. The factors affecting it are the medium access control protocol used and the length of the queue ahead of the current packet.

S is the service time – the time taken to transmit all the bits in the packet (proportional to the size of the packet). It depends on the data rate of the channel.

T is the propagation delay - the time taken for a single bit to travel to the destination. It is a characteristic of the channel used and is proportional to the distance between the source and the destination.

As we can see here, simulation of different topologies, channels and the medium access control protocols give rise to the different delays experienced by a packet.

In our simulator, the computing nodes send their messages out to a network which then inserts a certain delay (characteristic of the network type) and then forwards the message to the destination.

3.1.5 Fault Injection

In the current model, only the computing nodes can get the faults. The user can specify the faults in a variety of ways including: the Poisson rates for the fault arrival and repair, a table of fault arrival events and repair etc. It can be extended to include other possibilities.

There is a fault generator (we will see later on its implementation) which computes the fault arrival/repair events and send them out to the computing nodes.

When the fault arrival event comes up, the nodes simulate the fault by stopping the current task and also clearing further events from the event queue. Until the fault is cleared and some recovery action has been performed on the node, it will not respond further.

3.2 Distributed Simulation Issues

The test bed is a event-driven simulator and hence the simulation proceeds by executing events from an event queue ordered by the time of their occurrence. During the execution of an event, the state of the system changes and new events might also be created. These events are then inserted into the event queue to be executed later. In a distributed simulation, there are multiple such event queues in the system. The key to success is to let the simulation proceed as efficiently as possible without violating the causality constraints [28].

Each of the nodes/network have a separate event queue. We now have to decide on how to let the simulation clock proceed. One possibility (and which we have followed

in our simulator) is to have a central clock that collects the next event time from all the event queues, computes the next event time for the entire system and then broadcasts this time to all. This ensures correctness so that the events proceed in an orderly fashion. The nodes execute the events that are supposed to occur at that instant. As part of this process, more events get generated and get inserted into the respective event queues. The clock collects the next event times from all the nodes and the process repeats.

Other possibilities include some kind of optimistic execution in which the simulation time proceeds in parallel until there is a interaction between nodes that might violate causality. It is more complex to implement but definitely improves the performance of the system as a whole.

3.3 RAMP Algorithm

When a fault has been detected on a slave node, there are a few possible recovery actions that can be done. Some of the most commonly used ones are

Retry Restart execution on the same node from a consistent state as recorded in the latest checkpoint.

Replace Use the latest checkpoint from the faulty node to start executing tasks on a spare node.

Disconnect Use the latest checkpoint from the faulty node to distribute the tasks running on that node to the other non-faulty nodes in the system.

Each of these actions can have different overheads in terms of the time needed to complete them. Their success also depends on the current state of the system, the workload, fault characteristics etc. Given all this information, the master node has to decide which action to take. The policy that the master uses could vary from a fixed action or something that gives certain weightage for all these parameters and then decides on a action.

RAMP is an example of a resource management algorithm that can be used. It is intended to suggest the most suitable recovery action that will minimize the probability of the system losing a critical deadline over the remaining mission time. For this, it takes into consideration the fault characteristics, workload, check-pointing interval etc.

To make the computation time of the algorithm feasible, a method called the Reduced State Space Markov Decision Process (RAMP) [31] was developed. This method reduces the system state space to a manageable level without significantly compromising precision. A dynamic programming technique [20] is then used to compute the optimal recovery action. This computation is done *a priori*, before the simulation is started. Given the reduced system space, the computation is done for all possible system configurations for the system mission time. The output of the algorithm consists of the set of actions to take for all possible system configurations and workload characteristics at a given point of time (with a desired resolution) in the mission.

Whenever a fault is detected and a recovery action needs to be chosen, the RAMP algorithm is consulted. Complete information on the current system state such as the

number of nodes that are alive and faulty, the remaining mission time etc are passed to the algorithm. The RAMP algorithm then suggests an appropriate recovery action after looking at these values and the precomputed alternatives.

3.4 Simulator Implementation

The design of the simulator is closely tied to the kind of systems that are to be modeled. We have seen earlier about the simulation model. The simulator models this system by having a set of processes that communicate with each other by means of a portable message passing library. There is a process to model each of the computing nodes, the network interconnect, central clock and the console. We will now take a look at each of these.

3.4.1 Platform

The simulator is intended to be run on multiple machines in a transparent manner. That way, we could add more machines for the simulation when we need more computational capacity. For this reason, PVM is chosen as platform on which to build the simulator.

PVM [8] is a portable message passing library that can run on multiple physical machines in a manner that is transparent to the applications. It uses the native message passing mechanism of the underlying machines (example UNIX sockets) to provide an abstract view of a single virtual machine.

Machines can be added to the PVM at will, thereby making it easy to increase the computing capacity. The processes that run on top of PVM can exchange messages

through well defined mechanisms and semantics. PVM is available on a wide variety of systems and the virtual machines can be made of multiple, heterogeneous machines. The simulator code is developed in C++.

This chosen platform has the following advantages

- Using a network of workstations for the simulator makes available a scalable computing platform. This will be helpful as the simulated system becomes more complicated and requires more computing power.
- Implementing the various components using the PVM processes facilitates distributing the load across machines evenly.
- PVM, implemented on a flavor of UNIX provides a portable message passing interface. So the simulator can easily be adapted to a wide variety of machines.
- C++ is a powerful object-oriented language which can be easily used to model the complex systems that are being simulated.
- Since C++ is a popular language, it will be easy for users to develop their own algorithms and integrate them with the simulator.

3.4.2 Implementation Model

There are two parts to the simulator implementation. How the actual system is simulated and how it is controlled from the users perspective. This can be modeled as shown in Figure 2.

The console acts as the controlling interface to the rest of the simulator. We can enumerate its functionality as follows

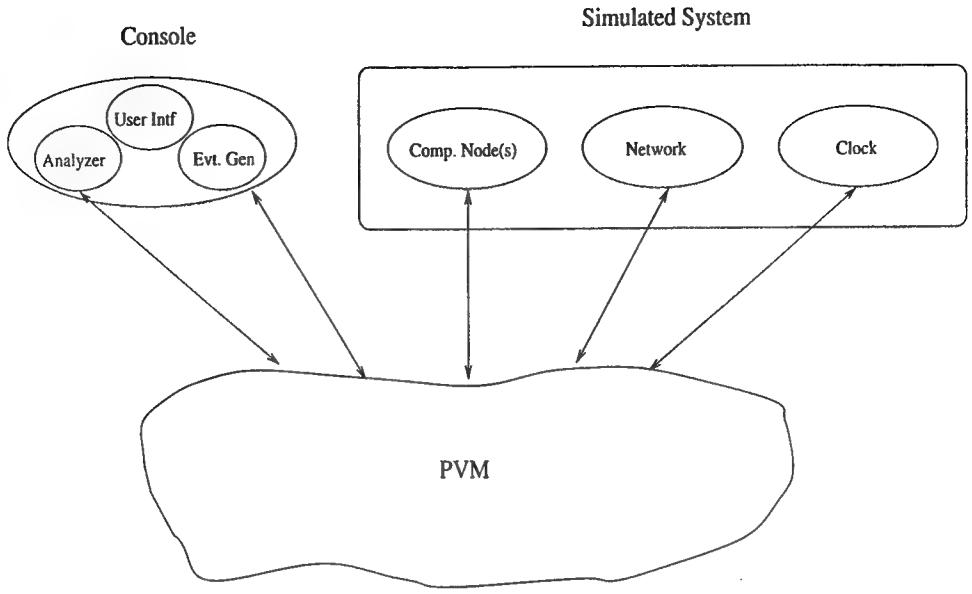


Figure 2: The Implementation Model

- Provides the GUI to help the user specify the system
- Provides the controls for the simulator
- Collects information about the task description, fault arrival patterns etc.
- Generates the fault arrival/repair events and send them out to the nodes
- Spawns the required processes
- Collects the information about the current state of the system
- Provides the status to the user via the GUI.

3.4.3 Other Simulator Entities

Virtual Nodes

Each virtual node represents a computing element in which the tasks run and exchange messages. The nodes run in a master-slave configuration. One of the nodes acts as the Master and does the control tasks such as task allocation, fault detection, fault recovery etc. Usually, the node with the highest id acts as the Master. If this node goes faulty, then some other node takes over. The slave nodes schedule and execute tasks which are allocated by the master.

Facilities are provided for plugging in the various algorithms that the nodes use such as the task allocation, scheduling etc. In the current version of the simulator, some of these algorithms are already in place and the future ones can be incorporated with minimum effort.

Virtual Network

The nodes can be connected via broadcast or point-to-point links. A variety of protocols such as FDDI and IEEE 802.5 Token Ring can be used on the broadcast links. In the case of point-to-point links, the nodes maintain routing tables to forward messages to the appropriate destination. A single process simulates the network connect in the system. It inserts the appropriate delay into the message passing depending on the protocol that is simulated in the network. We will now take a look at some special considerations for the simulation of individual networks.

Token Ring

Efficiency is an important issue in simulating the network protocols. In a token passing protocol, the token keeps on circulating even where is no message in the network. Each of these token passing events have to be simulated and hence it slows down the system considerably. The alternative is to stop the token passing when there are no messages in the system and restart it only when a new message comes into the system. To implement this, we record the position and the time when the token was last seen and regenerate the token at the appropriate place when a new message arrives.

The parameters of the network such as the operating speed, the number of nodes, token length, node latency, the token holding time etc, can be varied to reflect the network under consideration.

FDDI

FDDI medium access control [15] is similar to the token Ring in that it also depends on token passing in a ring. However it follows a timed token protocol which allows each node to reserve a portion of the available bandwidth to send Synchronous data (which is the real-time traffic) at periodic intervals. Each node can calculate the minimum amount of data that it needs to put out every P_i time units and send this information to the network. The network can calculate the fraction of the bandwidth needed for each node and assigns them to the nodes. This amount of bandwidth is guaranteed for each node over the requested period. Any unused bandwidth can be used by Asynchronous data which is the non-real time traffic.

For a more detailed discussion on the mechanism of bandwidth allocation, and the cycle time properties of the FDDI algorithm, see [16]. The efficiency considerations that occurred for the token ring also applies in this case and we follow the same approach.

Central Clock

The virtual nodes and network have their own local version of virtual time and run independently of others except when there is a need for interaction. They must satisfy causality constraints [17] to ensure correctness of the simulation.

The RAPIDS simulator uses a variation of the Breathing Time Buckets technique [28]. Each process in the simulator has a notion of the Local Event Horizon, which is the time it can proceed to, without violating causality. The central clock maintains the Global Event Horizon which is always the minimum of the Local Event Horizons, and broadcasts this as the current time of the entire system.

3.4.4 Fault Injection

As we have seen earlier in the implementation model, the console has an event generator which is in charge of generating the fault arrival and repair events. The fault injector reads the fault generation information in one of two possible formats.

- Poisson fault arrival, repair rates
- Table of fault arrival times and repair times

Looking at this information for each of the computing nodes, it generates the two types of events (fault arrival and repair). By not sending out the fault repair event

for a node, the permanent faults can be simulated. A predefined number of these events are generated and sent out to the concerned nodes. These nodes insert the events into their respective event queues and simulate the events at the appropriate time.

These events are periodically replenished whenever the number of events falls below a low-water mark. Since the nodes send their event information back to the console, a precise estimate can be made about the number of events still to be executed.

3.4.5 Fault Detection and Recovery

Fault on the Slave Nodes

The master node is responsible for detecting the failure of a slave, and invoking the appropriate recovery actions. Each fault-free slave node periodically records its state in a checkpoint. It also periodically sends an "I am alive" message to the master. The master node periodically examines these slave messages to decide if any of them are faulty. The period of the alive messages and the master invoking the maintenance functions are design parameters that can be specified.

The master node maintains the following information about the system to help in allocation and reassignment of tasks etc: The utilization of each node, whether the node is currently alive, whether the node is a spare, whether the node has sent an Alive message recently, the recommended recovery action for the node.

On detecting a fault, the master invokes a recovery algorithm to decide on the appropriate recovery action.

The three recovery actions have different penalties in terms of the time taken to perform them. The user can specify these values before the start of the simulation. User can also specify the algorithm to be used to generate the recovery actions during the mission time. This can be one of the following: a fixed action, a heuristic or an optimal recovery policy algorithm similar to the RAMP algorithm.

Interfacing with the RAMP algorithm

The RAMP algorithm precomputes and outputs the set of actions to take for all possible system configurations and workload characteristics at a given point of time (with a desired resolution) in the mission. The selection of a particular action depends on the fault characteristics (transient and permanent fault rates, transient fault duration), the given workload, the checkpointing interval and the remaining system mission time.

Some of the parameters for the RAMP algorithm are needed at the beginning, to generate the tables. These are either taken from the user, or estimated by doing a pre-simulation on the user inputs. Others are run-time parameters indicating the current time in the mission, number of nodes that are currently up, the utilization of these nodes etc. These are calculated at runtime and passed onto the algorithm. The algorithm uses this information to select the appropriate recovery action and gives this information to the simulator to implement.

Fault on the Master Node

The master node also sends an "I am alive" message to the slaves and the slaves monitor this message. The non-receipt of this message indicates that the master has gone faulty, and a new master has to be elected.

The slaves follow a 'Bully' algorithm for the election in which the slave with the highest id becomes the new master. As soon as the failure is detected by a node, it sends an election message to the other nodes that have a higher id. The nodes getting the election message respond with an acknowledgement whereupon the original node(s) gives up and waits. If no node sends back an acknowledgement, then the node that initiated the election wins and becomes the new master. The node that sends out the acknowledgement has the responsibility of finding the new master and so it initiates another election (if it hasn't done so yet). Ultimately, the node with the highest id (that is currently alive) is found and becomes the new master.

CHAPTER 4

SIMULATION ANALYSIS

Real-time systems have to be carefully validated before they are put into operation. Specifically, we need to know the probability that the system will not fail over the mission time. Here we also need to be specific about what we mean by "failure of the system". This chapter looks closely at this and tries to arrive at a framework to estimate it.

4.1 Reliability

When we have a mission-oriented system, we would like to know whether it will fail over its mission time. Reliability is the probability that the system will not fail over its mission time. Unreliability is the complementary probability (the probability that the system will fail over its mission time). To find this, we need to define what exactly we mean by 'failure' in our context. We will now try to define this in the context of a hard real-time system that we intend to model.

In a hard real time system we can have critical and non-critical tasks. Critical tasks need to be executed before their deadlines and if these deadlines are not met, catastrophes can occur. Non-critical tasks are not critical to the application. However,

they do deal with time-varying data and hence are useless if not completed within their deadline. The objective of a hard real time system is to meet all the critical deadlines and if possible, the non-critical deadlines too.

The Critical tasks are often executed at a higher frequency than is absolutely necessary. This constitutes time redundancy and ensures that one successful computation every n iterations of the critical task is sufficient to keep the system alive. The actual value of n will depend on the application, the nature of the task and it's frequency.

We define the system failure state as the state where it failed to meet the required number of iterations out of the n iterations. Even if a single critical task does not meet its deadline, then it is deemed a failure of the system itself. For each critical task, we can have a moving window and check whether the system is able to meet the specified number of deadlines among the iterations.

In order to measure the unreliability, we can run the system for its mission time and see whether it fails. This experiment is repeated many times and the percentage of times the system failed gives the estimate of the unreliability of the system.

4.2 Normal Simulation for Estimating Reliability

A simulation study is usually undertaken to determine the reliability of the system. In cases where the value of the reliability is very close to 1, it is usually better to estimate the unreliability of the system which is $1 - reliability$. Lets call the unreliability of the system as θ which is the parameter to be estimated from a simulation

study. The system is simulated for the mission time to see if it fails. The output value X_i is a 1 or 0 depending on whether the system failed or not. The simulation is repeated to get more such data points. The mean of these data points $\overline{X}$ provides an estimation of θ.

By the central limit theorem, a $100(1-\alpha)\%$ confidence interval for θ is approximately $\overline{X} \pm z_{\alpha/2} S/\sqrt{n}$ where S is the observed standard deviation and $z_{\alpha/2}$ is defined by the equation

$$\alpha/2 = P(N(0,1) \geq z_{\alpha/2})$$

$N(0,1)$ denotes a normally distributed random variable with mean 0 and variance 1. By definition,

$$X_i = \begin{cases} 1 & \text{with probability } \theta, \text{ and} \\ 0 & \text{with probability } 1-\theta \end{cases}$$

and hence the variance of X_i is given by

$$Var(X_i) = \theta(1-\theta)$$

Suppose we wish to estimate θ to within $\pm 10\%$ (about two significant digits of accuracy), i.e, we want the relative half-width of say, a 90% confidence interval for θ to be less than 0.1, that implies :

$$1.282\sqrt{\theta(1-\theta)/n}/\overline{X} \leq 0.1$$

$$n \approx 1.282^2 * 100 * (1-\theta)/\theta$$

From this, we can see that the required sample size is proportional to $1/\theta$. The smaller θ is, the larger the sample size must be. For example, if θ is of the order of 10^{-6}, a sample size of the order of 10^8 will be needed.

For the kind of systems that we are considering, the probability of failure is very low for the individual components and the unreliability of the complete system is very low. Therefore a very large number of samples is required to estimate this parameter to a reasonable level of accuracy.

CHAPTER 5

IMPLEMENTING IMPORTANCE SAMPLING

In the earlier chapters we have discussed the basics of importance sampling, how the simulator test bed is implemented and how the simulation outputs can be analyzed to estimate the reliability of the system.

This chapter talks about how all these are put together to actually implement the desired solution and how exactly we validate the implementation. It also takes a look at the parameters that affect the simulation output and provides some guidelines on how to choose them.

5.1 Outline

The implementation model of the system is as shown in Figure 3. To implement importance sampling in this model, we should not alter the simulated system, we only need to modify the generation of the fault arrival/repair events and the way the reports are analyzed.. We intend to measure the unreliability of the system by repeating the simulations to get individual samples. The algorithm to be followed can be summarized as

- Initialize the weight of the simulation output to 1.

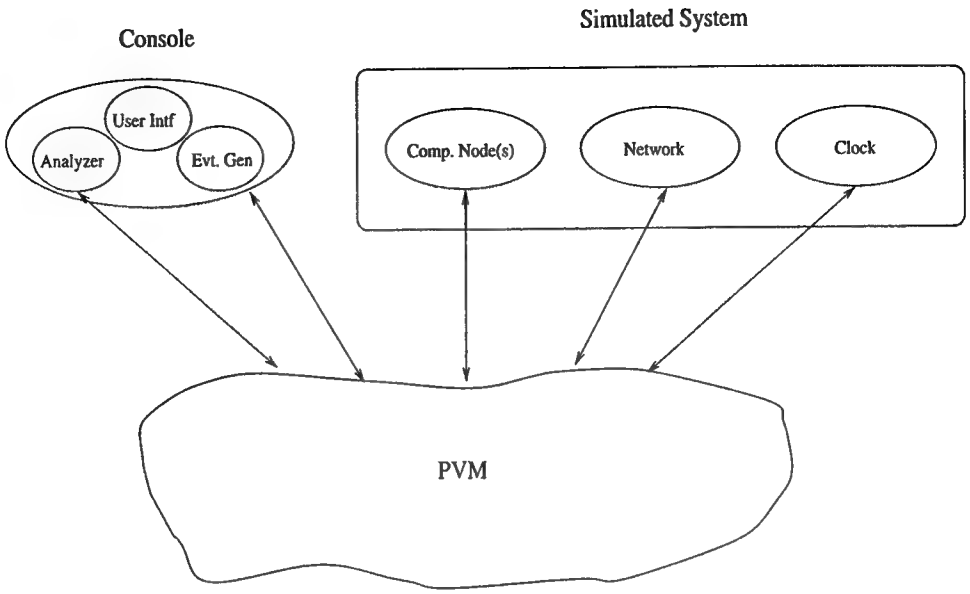


Figure 3: The Implementation Model

- Use the heuristics of "forcing" and "failure biasing" to generate the fault arrival/repair events.
- Send these events to the individual components and replenish them periodically.
- Monitor the reports from the simulated system.
- Update the simulation weight whenever a "change of measure" is performed.
- Output either the simulation weight, when the system fails before the end of the mission or zero, when the mission ends with the system still functioning. This forms a single sample of the simulation run.

The logical place to implement importance sampling is in the console. To be more precise, we can implement this in the event generator and the analyzer. The event generator has the following responsibilities

- Decide the time of the next system state transition. Implement "forcing" to accelerate the state changes.
- Decide whether the next transition is a fault arrival or repair. Implement "failure biasing" to push the system towards more component faults.
- Calculate the likelihood ratios associated with each "change of measure" and store this value along with the time it is supposed to happen.

The analyzer has the following responsibilities

- Receive the reports from the simulated system.
- If it corresponds to one of the above mentioned "change of measure", update the current simulation weight.
- If the system fails within the mission time, set the simulation output to the current value of the likelihood ratio else set the simulation output to zero.

We will see later how each of these functions are implemented in the modules.

Theoretical Formulations

The equations governing system failure are constructed from two probability density functions [18]. Let

$f(t|t', k') \equiv$ Probability density that the system will make a state transition at

t given that it is at state k' at time $t'(t' \leq t)$ and

$q(k|k')$ $\equiv$ Probability that the system will enter state k, following a transition out of state k'.

Now, let $\psi_k^n(t)$ be the probability density that the system arrives in state k at time t after the n^{th} transition. Then, the probability density for arrival in state k is:

$$\psi_k(t) = \sum_{n=0}^{\infty} \psi_k^n(t).$$

A recursive relation for the states for which $n > 0$ follows immediately from the definitions of the probability densities:

$$\psi_k^n(t) = \sum_{k'} q(k|k') \int_0^t dt' f(t|t', k') \psi_{k'}^{n-1}(t')$$

Suppose we consider a modified or "biased" random walk where the probability density $f(t|t', k')$ and the discrete probabilities $q(k|k')$, both defined above, are replaced respectively by the modified distributions $\tilde{f}(t|t', k')$ and $\tilde{q}(k|k')$.

Similarly we can define $\tilde{\psi}_k^n(t)$ as the modified density that the system arrives at state k at time t after the n^{th} transition. We now require a relation between the earlier density and the new density function. To do this, we define a weight $w_k^n(t)$ such that

$$\psi_k^n(t) = w_k^n(t) \tilde{\psi}_k^n(t)$$

This weight $w_k^n(t)$ is the likelihood ratio associated with the current state. Substituting this in the earlier equations and simplifying we can get the recursive relation

$$w_k^n(t) = \frac{q(k|k') f(t|t', k')}{\tilde{q}(k|k') \tilde{f}(t|t', k')} w_{k'}^{n-1}(t')$$

The above is the key equation that we will need for the implementation. We can associate a weight with each random walk, which is initialized to one and then adjusted to correct for the bias at each sampling. The incremental likelihood ratio for each "change of measure" is given by

$$\frac{q(k|k')f(t|t',k')}{\tilde{q}(k|k')\tilde{f}(t|t',k')}$$

This has two parts, $q(k|k')/\tilde{q}(k|k')$ and $f(t|t',k')/\tilde{f}(t|t',k')$ each corresponding to the likelihood ratio due to the two heuristics of failure biasing and forcing respectively. Since these are independent, we can calculate the ratios due to each of them separately and then multiply them to give the likelihood ratio for the state transition.

The cumulative weight when the system enters a failed state is the sample value(X_i) for the run. If the system doesn't fail over the entire mission, the resultant sample value is 0.

5.2 Implementation

The major chunk of the work is implemented by the event (fault) generator. It generates the list of the fault arrival/repair events for all the computing nodes in the system. To do this, it maintains a set of variables for each node i in the system. The fault arrival rate is given by λ_i and the repair rate is given by μ_i.

λ is the total failure rate out of the current state and is equal to the sum of the failure rates of active nodes. μ is the total repair rate out of the current state and is equal to the sum of the repair rates of all the currently faulty nodes (but not

permanently faulty). Finally γ is the total transition rate out of the current state and is equal to the sum of λ and μ. As a node goes faulty or gets repaired the system state changes and these variables are updated.

5.2.1 Forced transitions

The fault generator module uses a random number generator to generate random numbers uniformly distributed between 0 and 1. To sample time intervals Δt between transitions, we can use the following equation

$$\int_{t'}^{t'\Delta t} dt f(t|t',k') = e^{-\gamma_{k'}\Delta t}$$

A random number ε is sampled from a uniform distribution $0 < \varepsilon \leq 1$, and set equal to the cumulative distribution on the left. Inverting the equation then yields

$$\Delta t = -\frac{1}{\gamma_{k'}} ln\varepsilon$$

where we have utilized the fact that ε and $1-\varepsilon$ have the same probability densities.

This value Δt is added to the present time t' to give the next transition time t. Usually the failure rate of the nodes is very small when compared to the repair rate (for transient faults). Therefore γ is small when no failed components are present. In such a case, the transition rate is boosted by taking

$$\begin{aligned} \tilde{f}(t|t',k') &= \frac{f(t|t',k')}{1-e^{-\gamma(T-t')}} \text{ for } t' \leq t \leq T, \\ &= 0 \text{ otherwise} \end{aligned}$$

where T is the mission time of the system.

This heuristic is applied only when γ is small, $\gamma(T - t') \ll 1$ and there is only a small chance that an additional transition will take place before the end of mission time T.

We then have to multiply the trial weight by

$$\frac{f(t|t', k')}{\tilde{f}(t|t', k')} = 1 - e^{-\gamma(T-t')}$$

This value is the partial likelihood ratio and is stored in a variable for use later.

5.2.2 Failure biasing

Once the next transition time is decided, we have to see whether it is going to be an additional node failure or a repair. Obviously if there are no failed nodes present, there cannot be a repair and hence the only possibility is a node failure. Similarly if all the nodes are faulty, repair is the only way to go. Barring these two extreme cases, we have to make some decision on the type of transition.

We would like to bias the transitions in such a way as to cause more failures and thus push the system towards system failure quicker than in the normal case. To recapitulate, system failure is defined as the state where the system is not able to meet all its critical task deadlines.

As more and more nodes go faulty, the load gets redistributed among the working nodes thereby increasing their loads. As the loads keep building there comes a point when the scheduler cannot meet all the task deadlines and hence the system will start missing some of the deadlines. If this condition persists (i.e., the faulty nodes remain down long enough) critical task deadlines will be missed and the system will

fail. Also, when the tasks are being moved around, there is an overhead involved in terms of time and deadlines could be missed even though enough computing capacity is available somewhere in the system.

Suppose for a particular system state k' we divide all possible transitions $k' \rightarrow k$ into two classes, the transition $k \in F$ corresponds to an additional component failure , while $k \in R$ corresponds to an additional component repair. Hence for non absorbing states

$$\sum_{k \in F} q(k|k') + \sum_{k \in R} q(k|k') = 1$$

We choose a fraction called the failure bias which indicates the percentage of the transitions that result in an additional fault (This affects the dynamics of the sample output and we will look at it in more detail later). Let us assume that we choose a value ϕ, then,

$$\sum_{k \in F} \tilde{q}(k|k') = \phi$$

and hence,

$$\sum_{k \in R} \tilde{q}(k|k') = 1 - \phi$$

We generate a random number ε' and compare it with ϕ. If it is smaller, then the next transition will be an additional fault. Otherwise, it will be the repair.

We choose to implement one of the variants of the failure biasing called "balanced failure biasing" as we have seen earlier. To decide on the exact node getting the fault, we maintain an ordered list of all the eligible nodes (lets say n) and generate a random number i uniformly distributed between 0 and $n - 1$. This i is the index of

the node getting the fault. The unbiasing fraction is then given by

$$\frac{q(k|k')}{\tilde{q}(k|k')} = \frac{n\lambda_i}{\phi\gamma}$$

If it is a repair, we decide the repaired node by looking at the repair rate of the individual faulty nodes (excluding those permanently faulty). For this we order the eligible nodes and determine the particular node i by using the following formula

$$\sum_{i'=1}^{i} \mu_{i'} < \frac{\mu(\varepsilon' - \phi)}{1 - \phi} \leq \sum_{i'=1}^{i+1} \mu_{i'}$$

The unbiasing fraction is then given by

$$\frac{q(k|k')}{\tilde{q}(k|k')} = (1 - \phi)^{-1} \frac{\mu}{\gamma}$$

5.2.3 Analysis

As we have seen in the last two subsections, we can implement the two techniques of forcing and failure biasing. In each case we calculate the unbiasing fraction also. The product of these two gives the likelihood ratio for that transition.

This value is associated with the transition event and is stored in a list. The overall likelihood ratio for the simulation run is initialized to one. As the event actually happens in the system, the fractional value is taken out of the list and multiplied with the running product to give the current value of the likelihood ratio.

The nodes report all the events happening in the system to the console. If the system is not able to meet its critical task deadlines, this constitutes system failure

and the current value of the likelihood ratio is the sample output. If the system does not fail over the mission time, then the sample value is a zero.

The simulator maintains a file to hold the sample outputs and another to store the seed for the random generator. Since the random generator is a crucial component of the simulator, its seed has to be maintained over the sample runs to make sure the numbers are uniformly distributed and also the simulation can restart from the last paused point.

5.3 Expected Behaviour of Importance Sampling

Before we carry out the validation of the implementation, it is imperative for us to understand the expected behaviour of the importance sampling scheme.

Importance Sampling is a heuristic, designed to speed up the occurence of rare events. We wonder whether it can perform uniformly well in all regions of unreliability values. Intuitively we know that there should be some range of the unreliability values beyond which the usage of the normal simulation will be better than the importance sampling.

We will use the 'Relative Error' (RE) defined as follows, as the performance measure of the estimation scheme (normal simulation and importance sampling).

$$RE = \frac{z_{\alpha/2} S}{\sqrt{n}\theta}$$

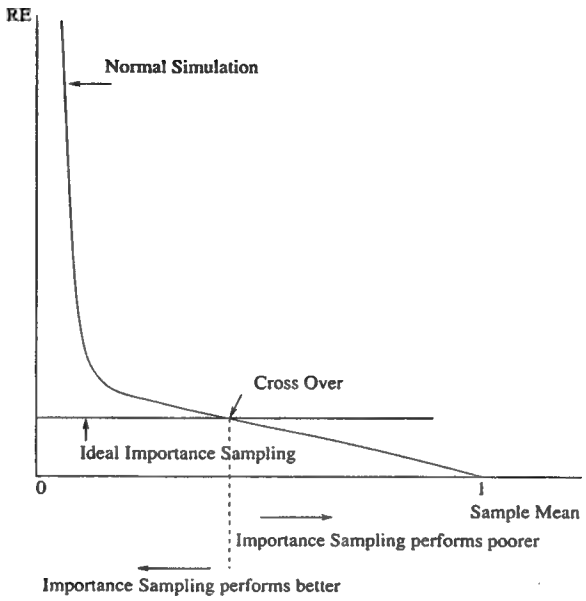


Figure 4: RE of Normal Simulation and Importance Sampling

For the case of the normal simulation, this becomes

$$RE = \frac{z_{\alpha/2}\sqrt{\theta(1-\theta)}}{\sqrt{n}\theta}$$

for a fixed number of samples, as the failure event becomes rarer (i.e., $\theta \to 0$) the $RE \approx z_{\alpha/2}/\sqrt{n\theta}$ becomes unbounded. Therefore, to obtain precise estimates, we need very large n.

For importance sampling, Shahabuddin [25] notes that the elements of the modified transition matrix should be independent of the 'rarity' (a parameter that he defines to reflect the highly reliable nature of the components) of the components. This is sort of an 'ideal' importance sampling scheme which will always lead to the

bounded RE property. He also proves in this case, the RE will have a form

$$RE = \frac{z_{\alpha/2}}{\sqrt{n}} \frac{\sqrt{a_2 + o(1)}}{(a_0 + o(1))}$$

where a_2 and a_0 are positive constants depending on the implementation.

For a fixed number of samples, we can represent the behaviour of RE as shown in Figure 4, for the cases of normal simulation and importance sampling. This 'ideal' importance sampling scheme will be able to estimate the sample mean with a small, fixed number of samples. We also observe that there is a certain region of reliability values beyond which the normal simulation will be better than the importance sampling.

In practice, techniques such as 'Balanced Failure Biasing' approximate the beviour of this 'ideal' case. They achieve variance reduction (and hence the RE) by pushing the system towards more faults and hence reducing the reliability of the underlying system. They then unbias the sample output to get the correct value of the sample mean.

The RE offered by such heuristics may not exactly be a constant but could be a complicated function of the bias value and the type of the system under consideration. However as the reliability of the system keeps on decreasing, the bias value has to be kept arbitrarily low in order to maintain the RE close to that of the normal simulation. Hence beyond a point, the importance sampling is not very useful. It is important for us to experimentally determine this range of values so that the user can switch to the normal simulation instead of the importance sampling scheme.

Recapitulating the basic idea of the importance sampling scheme, the estimate of the sample mean is given by $E_{p'}[1_{\{X \in A\}} L(X)]$. Note that since we want to make the sample variance very low, we need to satisfy two things:

- More number of samples have to contribute to the result (this is achieved by pushing the system towards more faults and thereby increasing the chance of it failing and contributing a non-zero sample).

- The 'most likely' paths to failure have to be favored more since the likelihood ratio has to be well behaved (the proportion of the failure paths has to be proportional to the relative importance of these failure paths in the system under consideration)

In the case of the failure biasing heuristic, there is only one parameter with which to control both of these. If we increase the failure bias, more number of sample runs will result in system failure and hence it is good as far as the first condition is concerned. However failure bias also affects path that a sample run might take.

In the systems that we are interested in simulating, the most obvious failure path is the one where the system is unable to meet its critical task deadlines. This obviously depends on a variety of factors such as the number of computing nodes available, average task load on the computing nodes, the recovery overheads etc. Consider a rather crude example: let a system be composed of x computing nodes and to meet its critical task deadlines it needs atleast y of them. Increasing the failure bias has the effect of pushing the system towards complete breakdown (in other words, more of the nodes are pushed towards failure). In many cases this will be an overkill and

the likelihood ratio associated with these sample runs will be very low. Thus most of the sample runs will have a likelihood ratio that is very less and a few of them will have very large values. This effect causes the RE to blow up in cases where the bias value is very high.

Because of these conflicting effects associated with the bias value, each system might have a optimal bias value that pushes the system towards the failure path most of the time and still does not make it an overkill. Since it is not practical for us to locate this optimal value for each configuration, it is enough if we are able to guess a bias value that gives 'good' results over a range of systems.

5.4 Validation of the Model

To validate our implementation of importance sampling, we need to compare the reliability estimates with the ones generated using normal simulation (without any bias).

Choosing the Configurations

As we have seen earlier, normal simulation of the complex reliability models take a long time and need extremely large number of sample points to get reasonable confidence intervals. This is exacerbated by the fact that each of these simulation samples take a long time.

The simulator is primarily intended to model very complex system models and algorithms and hence the number of events to be simulated is very large. Because of this (and other factors such as the efficiency of the distributed simulation) each

simulation run takes on the order of minutes for mission times on the order of a couple of thousand units.

Because of these reasons, we need to be very careful in choosing the configurations that we use to validate the implementation. Some of the important considerations are

- We need to verify that the implementation works for a variety of configurations that can be modeled using the simulator
- We need to make sure that the accuracy of the results holds for configurations with various ranges of reliability values

Since we only aim to get ballpark estimates for comparing with the importance sampling scheme, we propose to repeat the simulations until we get the half width of the 90% confidence intervals to be around 30% of the sample mean.

We choose 4 different system configurations (with increasing reliability values) and estimate their unreliability without applying any bias. These configurations are as follows:

Conf1: Number of Nodes: 6

Network interconnect: Token Ring (4 Mbps)

Transient failure rates: $1.38 \times 10^{-3}(5/hr)$

Permanent failure rates: $2.77 \times 10^{-5}(1/hr)$

Mission time: 1000 units

Average load on the nodes: 0.4

Conf2: Number of Nodes: 7

Network interconnect: Fddi

Transient failure rates: $5.55 \times 10^{-4}(2/hr)$

Permanent failure rates: $5.55 \times 10^{-5}(0.2/hr)$

Mission time: 1500 units

Average load on the nodes: 0.3

Conf3: Number of Nodes: 8

Network interconnect: Rectangular mesh

Transient failure rates: $2.77 \times 10^{-4}(1/hr)$

Permanent failure rates: $2.77 \times 10^{-5}(0.1/hr)$

Mission time: 2000 units

Average load on the nodes: 0.25

Conf4: Number of Nodes: 8

Network interconnect: 3D hypercube

Transient failure rates: Half of the nodes had $1.38 \times 10^{-4}(0.5/hr)$

Other half had $2.77 \times 10^{-4}(1/hr)$

Permanent failure rates: $2.77 \times 10^{-5}(0.1/hr)$

Mission time: 2000 units

Average load on the nodes: 0.25

The repair rate for all of the configurations was 0.277 or (1000 /hr). This is to make sure that the nodes come out of the transient faults quickly. The sample output values are binary: If the system failed before the end of the mission time, the sample

Table 1: Normal Simulation

Configuration	Sample Mean	Sample Variance	Number of Samples	Conf. Interval
Conf1	0.0424	0.04062	2500	12.2%
Conf2	0.0032	0.0031904	5000	32%
Conf3	0.0007	0.000699	20000	34.25%
Conf4	0.0003	0.000299	40000	37%

Table 2: Importance Sampling with Bias 0.3

Configuration	Sample Mean	Sample Variance	Number of Samples	Conf. Interval
Conf1	0.0236	0.052	2600	24.29%
Conf2	0.00342	9.9×10^{-4}	1400	31.6%
Conf3	0.0006462	5.65×10^{-5}	1938	33.88%
Conf4	0.0002977	1.74×10^{-5}	2430	36.44%

value is 1. If the system did not fail before the end of the mission time, the sample value 0. This is repeated n times and the sample mean, variance and the confidence intervals are calculated as discussed in the last chapter.

Results

We run the simulations for the above mentioned configurations and the results are tabulated as in Table 1.

We then repeat the experiments with the importance sampling and a nominal bias of 30% (or 0.3). The results are as tabulated in Table 2. Comparing this with Table 1, we can observe the following.

For configuration 1, where the unreliability is quite high, the importance sampling did not provide any improvement. In fact, the importance sampling estimates

performed poorly. It shows clearly that importance sampling is not meant for such systems with a high unreliability. We might be able to reduce the bias value and see whether it performs better. But this may not offer much variance reduction over the normal simulation. This is the kind of limit that we disucussed in the last section. So when the unreliability is on the order of 10^{-2} or better, it is better to go with the normal simulation itself.

For the rest of the configurations, the importance sampling estimates agree quite closely to that of the normal simulations. In fact, the number of samples taken is much less when compared to the normal simulation. This is the kind of systems for which the importance sampling heuristic was designed.

Acceleration Factor

In order to see how well the importance sampling schemes have performed, we define a factor called 'Acceleration Factor' (AF). It is defined as the ratio of the number of samples of the normal simulation (Num_{norm}) to that of the importance sampling scheme (Num_{imp}) required to achieve the same amount of confidence intervals.

$$AF = \frac{Num_{norm}}{Num_{imp}}$$

For the above mentioned configurations, the acceleration factors are as shown in Table 3. As we can see, the acceleration is much better as the system reliability increases.

Table 3: Acceleration Factor

Configuration	Acceleration Factor (AF)
Conf1	undefined
Conf2	3.57
Conf3	10.32
Conf4	16.46

Comparison of the Time Taken

It is also important to compare the actual time taken (for the normal simulation and for the importance sampling) to get the desired results.

We will use Conf2 (where we only got a nominal value for the acceleration factor) as an example to look at the savings in time that is possible.

Normal simulation method took an average of around 8 mins (rounded off to the nearest minute) for each of the sample points. The importance sampling scheme took an average of 6 mins for each of its sample points. Hence the ratios of the total time taken can be calculated (similar to that of the acceleration factor)

$$\begin{aligned} \text{Ratio of the time taken} &= \frac{Time_{norm}}{Time_{imp}} \\ &= \frac{8 \times 5000}{6 \times 1400} \\ &= 4.76 \end{aligned}$$

As we can see here, the actual gain in the time saved is higher than that indicated by the Acceleration Factor. This gets better with configurations having long mission times.

When using Importance sampling, the events are forced to occur more rapidly and the system is forced to go into the failure mode faster. Because of this, the actual time taken for the simulation run gets shorter when compared to the normal simulation where most of the simulation runs take the entire mission time.

Hence we gain both in terms of the reduction in the number of samples required and also the time taken for each of these sample points.

5.5 Selecting the Bias Parameter

As we have seen earlier, failure bias is an important parameter that alters the dynamics of the sample output. If it is too high or too low, the variance of the sample output will be high. There is usually some optimal value associated with each system.

Since the simulator has to work with a large variety of systems with varying unreliabilities, we have to identify a nominal value of failure bias that will work well with most system configurations. Intuitively, if the sample variance is low, the estimates converge faster and we need fewer samples to get the desired confidence intervals.

For each of the above mentioned configurations we vary the failure bias from 0.2 to 0.6 in steps of 0.1 and observe how the variance of the sample outputs change accordingly. The results of this are tabulated in Table 4.

From the above table we observe that the optimal bias value (the one that produces the least sample variance) is different for each configuration and in general, the higher

Table 4: Sample variance for different failure bias values

Config (mean)	Bias = 0.2	Bias = 0.3	Bias = 0.4	Bias = 0.5	Bias = 0.6
Conf1 (0.0424)	0.048	0.052	0.055	0.0586	-
Conf2 (0.0032)	9.4×10^{-4}	9.9×10^{-4}	1.02×10^{-3}	1.37×10^{-3}	-
Conf3 (0.0007)	6.04×10^{-5}	5.65×10^{-5}	6.21×10^{-5}	6.27×10^{-5}	7.83×10^{-5}
Conf4 (0.0003)	3.2×10^{-5}	1.74×10^{-5}	1.69×10^{-5}	1.83×10^{-5}	2.92×10^{-5}
Conf5 (2.1×10^{-5})	4.7×10^{-7}	2.3×10^{-7}	1.9×10^{-7}	1.976×10^{-7}	2.53×10^{-7}
Conf6 (5.02×10^{-7})	-	3.07×10^{-9}	2.93×10^{-9}	2.64×10^{-9}	2.86×10^{-9}

the unreliability of the system, the higher the value of this optimal bias. This is in accordance with expected behaviour as mentioned in the last section.

For Conf1, the normal simulation seems to be the best choice. For all the failure bias values, the sample variance goes much above those for the normal simulation. Conf2 seems to have an optimal value around 0.2, Conf3 around 0.3, both Conf4 and Conf5 have an optimal value around 0.4 and Conf6 around 0.5. It seems to stabilize around 0.5 for configurations with higher reliabilities.

We can use these experiments to choose a failure bias when we want to estimate the reliability of a particular system. The optimal value of the bias will vary across different configurations.

A bias value of around 0.4 seems to work well for most of the configurations. If the 'guessed' unreliability of the system is quite high (on order of 10^{-3}), we are better off by choosing a bias of around $0.2 - 0.3$.

The simulator is intended to be used to systems whose unreliability values are quite low. So in most of the cases we can pick a bias value of around $0.4 - 0.5$ which works reasonably well over a range of values.

Table 5: Effect of the Transient Failure Rates on the Unreliability

Transient Failure Rates /hr	Unreliability
1	6.4×10^{-4}
0.8	4.8×10^{-4}
0.5	1.5×10^{-4}
0.2	9.7×10^{-5}
0.1	5.3×10^{-5}

5.6 Some Typical Usages

Once we have validated the implementation and understand how to choose the failure bias, we are ready to use this simulator tool.

5.6.1 Varying the Transient Failure Rates

We will now use the simulator to check how the reliability of a system varies with the change in the failure rates of the individual components. For this purpose, we can take one of the configurations and alter the transient failure rates to observe the corresponding change in the reliability of the system.

We take the Conf3 and estimate its unreliability for various values of the transient failure rates. A failure bias of 0.3 was employed throughout the experiment. The results are tabulated in Table 5.

Here we observe that the unreliability of the system decreases gradually as the failure rates of the individual nodes decrease.

Table 6: Comparison of the Recovery Policies

Configuration	Fixed Recovery Action	RAMP Algorithm
Conf2	0.00342	0.00336
Conf3	0.0006462	0.0002841
Conf4	0.0002977	0.0001736
Conf5	2.1×10^{-5}	1.3×10^{-5}

5.6.2 Comparing the Recovery Policies

As we have seen earlier, the RAMP algorithm [30] suggests the optimal recovery action to be used whenever there is a fault in the system and a decision has to be made (regarding the choice of the recovery actions). Even though it is theoretically proved to work well, it will definitely help to know how well it performs for a typical system of our choice.

Since the simulator provides a control on choosing the recovery policy to be used for the system, we can proceed by running two experiments. We can do one set of simulation runs with a fixed recovery action and another with the RAMP algorithm and compare the results.

A Fixed Recovery Action

The three basic recovery actions are Retry, Replace and Disconnect. They had the overheads of 1, 2, and 3 units of time respectively. Based on these numbers an intuitive fixed recovery action can be formulated such as the following:

- When the node fails, try a Retry first.
- If the Retry failed, then try to Replace the faulty node by a spare node.

- If a spare does not exist, then as a final resort, Disconnect the faulty node and distribute its load to the other active nodes.

We used such a fixed recovery action in the configurations Conf2, Conf3, Conf4 and Conf5 mentioned earlier. The unreliability estimates of these configurations are shown in the Table 4. The failure bias used was 0.3

Comparison with RAMP Algorithm

Next, we run the same configuration, replacing only the fixed recovery policy with the RAMP algorithm to estimate the new unreliability. This experiment can be repeated for other configurations and the results are compared in Table 6.

From this we can see clearly that the RAMP algorithm performs better than the intuitive fixed recovery action for all of the considered configurations.

CHAPTER 6

CONCLUSIONS

This thesis work consisted of the implementation of an efficient variance reduction technique called importance sampling in a simulator testbed.

- The implementation of this technique is validated by running a series of simulations for different configurations and comparing the results with that of a normal simulation.
- The effect of the failure bias on the sample variance has been investigated to provide some guidelines in choosing a good failure bias probability for a given system.
- The tool is used to
  - Observe the change in the unreliability of the system with the change in the transient fault rates
  - Demonstrate that using an optimal failure recovery algorithm such as RAMP can significantly improve the reliability of a real-time system.

A couple of improvements can be done to the implementation to improve the usability of the tool.

- **The efficiency of the simulation.** Since the simulator was designed to model very complex interactions between the hardware and the algorithms, the number of events to be simulated is very large. A pessimistic algorithm is used to coordinate the different clocks in the nodes. Because of these reasons, each simulation run takes a very long time. It might be very beneficial if the events are studied closely to discover the potential bottlenecks.

- **Choice of the failure bias.** In the present model, it is left to the user to make a good guess of the failure bias. It might be possible to make the simulator learn from the past history and adapt the value of the failure bias.

Bibliography

[1] S. Andradottir, D. Heyman and T. Ott, "On the Choice of Alternative Measures in Importance Sampling with Markov Chains," *Operations Research* vol.43, no.3, pp.509-519 1995.

[2] M. Berg and I. Koren, "On Switching Policies for Modular Fault-Tolerant Computing Systems," *IEEE Trans. Computers,* Vol. C-36, pp. 1052-1062, Sept. 1987.

[3] M.Boyd and S.Bavuso, "Simulation Modeling for Long Duration Spacecraft Control Systems," *1993 Proc. Annual Reliability and Maintainability Symposium,"* pp 106-113 1993.

[4] J.Carrasco, "Failure distance based simulation of repairable fault-tolerant systems," *Proc. of 5th International Conf. on Modeling Techniques and Tools for Computer Performance Evaluation,* 1991, pp 337-351.

[5] J.Carrasco, "Efficient Transient Simulation of Failure/Repair Markovian Models," *Proc. of 10th Symposium on Reliable and Distributed Computing,* IEEE Computer Society Press, 1991, pp 152-161.

[6] J.Dugan, K.Trivedi, M.Smotherman, R.Geist, "The Hybrid Automated Reliability Predictor," *AIAA Journal of Guidance, Control, and Dynamics,* 1986, vol. 9.

[7] P.L'Ecuyer, "Efficiency Improvement and Variance Reduction," *Proc. of the 1994 Winter Simulation Conf.* pp. 122-132 1994.

[8] Al Geist, A. Beguelin, J. Dongarra, W. Jiang, R.Manchek, V.Sunderam, *PVM: Parallel Virtual Machine,* MIT Press, 1994.

[9] P.Glynn and D.Iglehart, "Importance Sampling for Stochastic Simulations," *Management Science*, vol. 35, no. 11, pp. 1367-1393, 1989.

[10] A.Goyal and S.S.Lavenberg, "Modeling and analysis of computer system availability," *IBM J. Res. Develop.*, vol.31 pp.651-664, 1987.

[11] A.Goyal, P.Heidelberger, P.Shahabuddin, "Measure Specific Dynamic Importance Sampling for Availability Simulations," *1987 Winter Simulation Conference Proceedings,* IEEE Press 1987.

[12] A.Goyal, P.Shahabuddin, P.Heidelberger, V.F.Nicola and P.W.Glynn, "A Unified Framework for Simulating Markovian Models of Highly Dependable Systems," *IEEE Transactions on Computers,* vol.41 no.1 pp. 36-51, 1992.

[13] J.M.Hammersley and D.C.Handscomb, *Monte Carlo Methods*, Meuthen, London, 1964.

[14] P. Heidelberger, "Fast Simulation of Rare Events in Queueing and Reliability Models," *ACM Transactions on Modeling and Computer Simulation* Vol. 5, No. 1, 1995.

[15] R. Jain, *FDDI Handbook,* Addison-Wesley, 1994 .

[16] C. M. Krishna and K. G. Shin, *Real-Time Systems,* McGraw-Hill, 1997 .

[17] L. Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System," *Communications of the ACM,* Volume 21, 7, 1978.

[18] E. E. Lewis and F. Bohm, "Monte Carlo simulation of Markov unreliability models," *Nuclear Engineering and Design*, Vol. 77, 1984.

[19] M.Nakayama, "A Characterization of the simple failure biasing method for simulations of highly reliable Markovian Systems," *ACM Trans. Model. Comput. Simul.* vol. 4, no. 1, pp 52-88, 1994.

[20] M. L. Puterman, *Markov Decision Processes,* John Wiley & Sons Inc., 1994.

[21] S. M. Ross, *Applied Probability Models with Optimization Applications,* San Fransisco: Holden-Day, 1970.

[22] S. M. Ross, *Simulation,* Academic Press, 1997.

[23] P.Shahabuddin, V.Nicola, P.Heidelberger, A.Goyal and P.Glynn, "Variance Reduction in Mean Time to Failure Simulations," *1988 Winter Simulation Conference Proceedings,* IEEE Press, 1988.

[24] P.Shahabuddin, "Simulation and Analysis of Highly Reliable Systems," Ph.D. Thesis, Department of Operations Research, Stanford University, Palo Alto, California.

[25] P.Shahabuddin, "Simulation of Highly Reliable Markovian Systems," *Management Science,* vol. 40, pp 333-352, 1994.

[26] P.Shahabuddin and M.Nakayama "Estimation of reliability and its derivatives for large time horizons in Markovian systems", *1993 Winter Simulation Conference Proceedings,* IEEE Press, pp 491-499.

[27] W. Stallings, *Handbook of Computer-Communications Standards,* Howard W. Sams & Co., 1988.

[28] J.S. Steinman, "Breathing Time Warp," *Proceedings of the 1993 Workshop on Parallel and Distributed Simulation,* 1993.

[29] K. K. Toutireddy, "A Testbed for Fault Tolerant Real-Time Systems," *M.S. Thesis,* Univ. of Mass. Amherst, 1996.

[30] K. Yu, "RAMP and the Dynamic Recovery and Reconfiguration of a Distributed Real-Time System," *Ph.D. Thesis,* Univ. of Mass. Amherst, 1996.

[31] K. Yu and I. Koren, "Reliability Enhancement of Real-Time Multiprocessor Systems through Dynamic Reconfiguration," *Fault-Tolerant Parallel and Distributed Systems*, D. Pradhan and D. Avresky (Editors), pp. 161-168, IEEE Computer Society Press, Los Alamitos, CA, 1995.

*U.S. GOVERNMENT PRINTING OFFICE: 2001-610-055-10071

DISTRIBUTION LIST

addresses	number of copies
RALPH KOHLER AFRL/SNRT 26 ELECTRONIC PKY ROME NY 13441-4514	2
DEPT OF ELECTRICAL AND COMPUTER ENGINEERING UNIVERSITY OF MASSACHUSETTS AMHERST MA 01003	2
AFRL/IFOIL TECHNICAL LIBRARY 26 ELECTRONIC PKY ROME NY 13441-4514	1
ATTENTION: DTIC-OCC DEFENSE TECHNICAL INFO CENTER 8725 JOHN J. KINGMAN ROAD, STE 0944 FT. BELVOIR, VA 22060-6218	1
DEFENSE ADVANCED RESEARCH PROJECTS AGENCY 3701 NORTH FAIRFAX DRIVE ARLINGTON VA 22203-1714	1
AFRL/SNOR ATTN: FRAN SMITH 26 ELECTRONIC PKY ROME NY 13441-4514	1